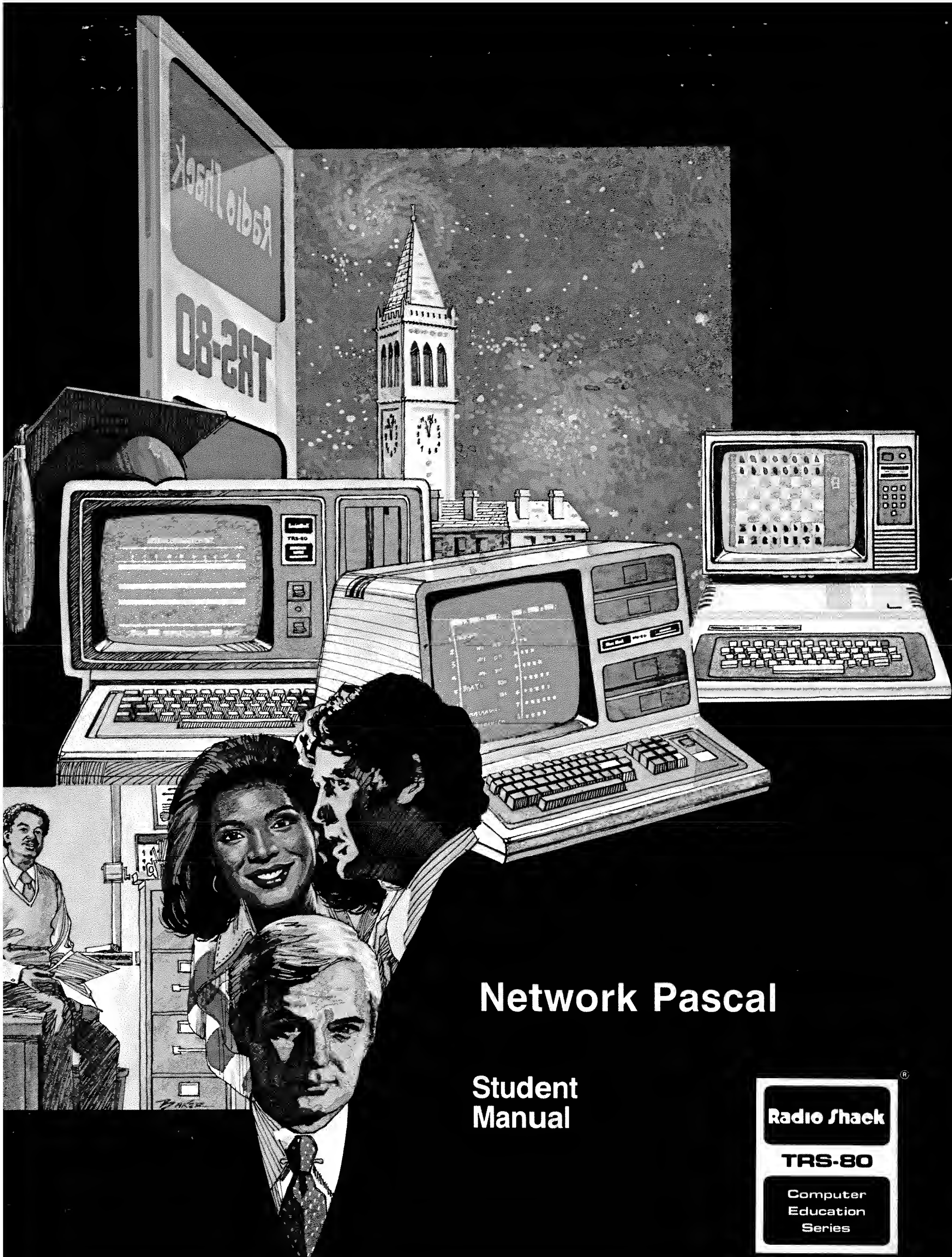


Network Pascal

Student Manual

Radio Shack®
TRS-80
Computer Education Series

Cat. No. 26-2740

THIS WARRANTY SUPERSEDES ALL PRIOR WARRANTIES

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this Radio Sback computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER tbat the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER's sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.

D. Except as provided herein, RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE." IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE."

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on one computer, subject to the following provisions:

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.

C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.

D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on one computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.

E. CUSTOMER is permitted to make additional copies of the Software only for backup or archival purposes or if additional copies are required in the operation of one computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the original CUSTOMER specific legal rights, and the original CUSTOMER may have other rights which vary from state to state.

NETWORK PASCAL

NOTICE TO PROGRAMMERS

By purchase of the Network Pascal package, you have obtained a license to duplicate TRSDOS and TRS-80 Pascal only as necessary for your own classroom use.

If you intend to sell or otherwise distribute Pascal applications programs developed using this system, you must follow the procedure below to avoid violation of this license and of copyright laws.

None of the Pascal programs provided with these diskettes may be reproduced for resale (including the TRSDOS operating system, the Pascal compiler, RUN program, and numerous auxiliary files).

TRS-80 Pascal produces an intermediate code which may be executed by the Pascal RUN utility. Alternately, the intermediate object code may be processed by the Pascal LINKLOAD utility to produce TRSDOS command files (/CMD extension files). These files contain TRS-80 Pascal support and will execute as stand-alone programs. They may be copied and sold freely with no royalty payments required. All programs that you sell in this manner must document the fact that they contain TRS-80 Pascal runtime support.

First Edition

INTRODUCTION

The Radio Shack® TRS-80® Network Pascal package contains three diskettes. The "Network Pascal" disk is designed for use on a Network 3 system with 48K Model III or Model 4 student stations and a 48K Host system with two or more disk drives. "Model III Pascal Disk One" and "Model III Pascal Disk Two" are designed for use on a 48K Model III or Model 4 stand-alone disk system.

The Network Pascal manual is divided into eight sections. The suggested approach is to study sections A and B first, since section A presents the fundamentals of using the programs in Network Pascal and section B presents the fundamentals of programming with the Pascal language. Sections C through E might be studied later, in the order in which they appear in this manual. Section F is for stand-alone system users. The GLOSSARY and INDEX should be referred to as needed. A summary of these sections follows:

SECTION A: NETWORK PASCAL BEGINNER'S GUIDE presents a hands-on demonstration of using Network Pascal. Following the steps in the BEGINNER'S GUIDE at a student station, you will enter, compile, and run a short Pascal program.

SECTION B: TUTORIAL is a ten-chapter introduction to the Pascal language, in a "textbook" format. Quizzes and activities allow you to test your understanding of Pascal as you work through this TUTORIAL.

SECTION C: PASCAL EDITOR REFERENCE GUIDE has two parts. Part One provides a complete overview of the Pascal editor, providing more details than were given in the BEGINNER'S GUIDE. Part Two describes the Model III Pascal editor program for stand-alone system users.

SECTION D: LANGUAGE REFERENCE GUIDE provides an in-depth look at the Pascal language, providing more details than were given in the TUTORIAL. Appendices to this section include a list of Pascal error codes (see page D-88) and the standard ASCII character set (see page D-93).

SECTION E: NETWORK PASCAL SYSTEM REFERENCE GUIDE provides an overview of the Network Pascal compiler (page E-2) and information on use of the linking loader (page E-5). Appendices to this section list the Pascal Procedure and Function Library, the String Function Library, and Random Access File Routines.

SECTION F: MODEL III PASCAL SYSTEM REFERENCE GUIDE presents a hands-on demonstration of using Model III Pascal on a stand-alone Model III or Model 4 disk system. Model III Pascal offers some Advanced Development Options, which are discussed in this section. Material in this section is arranged from least technical to most technical, and material toward the end of this section is intended for advanced programmers.

The **GLOSSARY** defines programming terms that may be unfamiliar to you.

The **INDEX** cross-references Pascal concepts with the pages on which they are discussed.

SECTION A: NETWORK PASCAL BEGINNER'S GUIDE

TABLE OF CONTENTS

NETWORK PASCAL BEGINNER'S GUIDE

1. DISKETTES IN THIS PACKAGE

The TRS-80 Network Pascal package contains three diskettes. The "Network Pascal" disk is for network use; "Model III Pascal Disk One" and "Model III Pascal Disk Two" are for optional use on a stand-alone Model III or 4 disk system.

This BEGINNER'S GUIDE focuses on the use of Pascal on a network system. The stand-alone diskettes are provided mainly to present the option of using a stand-alone system for some operations, and their use is discussed in the MODEL III PASCAL SYSTEM REFERENCE GUIDE (section F of this documentation).

2. SUGGESTED CONFIGURATIONS FOR NETWORK PASCAL

Several network configurations are possible. Appendix I of this BEGINNER'S GUIDE provides information to help the teacher decide on a configuration, based on the hardware available and the number of students using Network Pascal.

3. MAKING A BACKUP COPY OF EACH DISKETTE

Before the first time this Network Pascal package is used, a backup copy should be made of each diskette provided with the package. These backup copies should be used as working diskettes, and the original diskettes provided with the package should be stored in a safe place. Appendix II of this BEGINNER'S GUIDE provides complete instructions for making backup copies.

4. USING THE NETWORK PASCAL EDITOR/COMPILER PROGRAM

4.a. Ready the Network System

1. Set up the network system for the configuration you have chosen, by following the instructions in the Network or Hard Disk Network manual(s) that came with the network hardware and software you are using.

2. If you want to transfer Network Pascal files to the Hard Disk or to the Network Operating System diskette (as mentioned in Appendix I) and have not already done so, do it now.

3. Load the Host and Student Station operating software for the system you are using (floppy controlled or Hard Disk controlled.)

4.b. Load the Editor/Compiler

At each Student Station, when the **Network 3** prompt is showing, type **NETPCL** and press **ENTER.** When the editor/compiler is loaded, you'll see a list of options (the Main Menu).

4.c. View the Main Menu

Network Pascal's Main Menu displays the following options:

Network Pascal

E = Edit
P = Pascal Compile
M = Move files
D = Delete files
N = exit to Network
?

- **E** to **Edit** lets you write a program and make any needed corrections.
- **P** for **Pascal Compile** compiles your program (translates it to object code, a form that the computer can use).
- **M** to **Move files** lets you copy a program from Student Station memory to the Host disk, or from the Host disk to Student Station memory.
- **D** to **Delete** files lets you erase the source and/or object code currently in memory.
- **N** allows you to exit to the **Network** prompt.

4.d. Select an Option from the Menu

To select an option from this menu, you simply type that option's letter (in upper or lower case). For example: to begin editing, you'd type **E** or **e.** It is not necessary to press **ENTER** after typing an option selection. If you type a character other than the ones shown on the menu, the menu will be re-displayed.

When you select an option, Network Pascal may need to load a short overlay (a short portion of the program) from the network. While an overlay is being loaded, the message **Waiting** will be displayed. If the message is displayed for more than a few seconds, it means that the network is busy performing work for other students. When the network begins to transfer the overlay, the message **Loading** appears. Overlays are loaded only when they are needed; for example, when you change from editing to compiling.

A short demonstration designed to introduce these editor/compiler options begins on the next page.

5. DEMONSTRATION: USING NETWORK PASCAL

Following along with this section will help you gain familiarity with Network Pascal. In this demonstration, we'll write a short source program, compile it into an object file that can be run, save the program onto a disk at the host system, and then run the program.

NOTE: Since this demonstration requires a much greater than average amount of disk access, it is probably not a good idea to have many students completing these steps individually at their Student Stations. Students might instead complete these steps as a group at one or two Student Stations, or individuals might complete the demonstration at low-traffic times.

5.a. Entering a Program

Let's begin by using the edit option. With the Main Menu displayed, type **E**. Since there is not yet any text in memory, you'll see a relatively blank screen:

```
        EOF




             Ø LINES    46Ø8 BYTES FREE
```

The message at the bottom of the screen displays the total number of lines in the source file (in this case, zero) and the amount of memory (in "bytes," or characters) that is available for program text at that Student Station. The number of bytes free will partially depend on whether an object file is already stored in memory.

In order to enter text, we first need to insert at least one blank line. To insert a block of 16 blank lines, let's press the **CLEAR** key followed by the **E** key. (Or you could insert lines one at a time by pressing **SHIFT @** or, on Model 4, the **F1** key.)

By moving the cursor to any position and then typing, we can insert text into the file. If a character is overstruck, the old character will be replaced by the new one.

Now, let's enter a program into memory by typing the program lines shown below, exactly as shown. At the end of each line, press **ENTER** to start a new line. To correct a typing error, backspace using the left-arrow key. For a list of all of the special keys that can be used to move the cursor when editing a program, see page A - 8:

NOTE: The Alcor Pascal compiler does not distinguish between upper and lower case. To switch between all upper case and upper and lower case, hold down the **SHIFT** key and press **Ø**.

```
PROGRAM test;
BEGIN
WRITELN('* I am a Pascal wizard.');
END.
```

Once you have correctly entered the program, press the **CLEAR** key followed by the letter **C.** A pair of angle brackets <> in the bottom left corner of the screen indicate that you are now in command mode, where you can type a command for the computer to follow immediately (commands typed in command mode are not part of the program you are writing). Now type the command **exit** (or **EXIT**) and press **ENTER.** The Main Menu will be re-displayed.

5.b. Compiling the Program

Once the program has been entered into the Student Station's memory, the next step is to compile it. The Pascal compiler translates the source program into a form that the computer can understand. At the same time, it checks for errors in the source program and reports any errors it finds to the programmer.

To use the Pascal compiler, type **P** at the Main Menu. You'll see the message **Waiting**, and then the message **Loading**, as the Student Station waits for disk access and then loads a program overlay from the Host.

When the compiler's overlay is loaded, you'll see the message **Generate object (Y/N)?**. Let's assume that we think our program is correct, and we want to go ahead and have the computer generate object code (code that we can <u>run</u>) for the program. Type **Y.** You'll see the message **Print listing on printer?**. If you want a printed listing, press Y for "yes." (You should usually press N for "no.") The program will then be compiled, and a listing of the program will be sent to the screen or printer (whichever you specified). If you made any typing errors, you'll see a caret symbol (^) pointing to the line that contains the error, and an error code number will be given. At the end of the listing, the total number of errors is listed, and the **<Press any key to continue>** prompt is displayed.

If your total number of errors is Ø, you have successfully completed the compiling process, and you can now press any key to return to the Main Menu. Skip to **Moving Your Program** on page A - 5.

If any errors are listed, you'll need to correct the errors and then compile the program again.

5.c. Correcting errors

Since our demonstration program is short, errors should be fairly easy to find. Before exiting the compiler, look closely at the lines where errors are shown. If you see one error right away, but don't see any later errors, don't be surprised. One error near the beginning of a program can cause the computer to perceive errors in later lines that would otherwise be correct. (For example, spelling "**BEGIN**" as "**PEGIN**" causes a total of 3 errors!)

To correct errors, you'll need to get back into the editor program. To exit the compiler, press **ENTER.** At the Main Menu, then press **E.** Retype any incorrect characters.

Exit the editor program, try compiling the program again, and repeat the process as many times as necessary until the program is correct.

Hint: When looking for errors, don't overlook punctuation. Punctuation is very important in Pascal programs, but is easy for beginners to overlook!

5.d. Moving Your Program

Before you can run your Pascal program, you need to move it to a disk at the Host system. Moving the object code will allow you to run the program. Moving the source code will allow you to do future editing on the program. For this demonstration, let's move both. (If you exit Network Pascal without moving the source and/or object code to disk, whatever is not moved will be erased from the computer's memory and lost.)

At the Main Menu, press **M** to move files. Let's move the source program first. At the message **Source or Object (S/O):** type **S** for source. At the message **Read from or Write to disk:** type **W** to write to disk.

At the message **File name:**, you'll need to enter a unique name (a name that is different from the name anyone else in the class is using). Your filename can contain three parts:

- An alphanumeric name of up to eight characters, beginning with a letter. Example: **MYFILE**

- An extension of up to three alphanumeric characters, beginning with a letter. Example: **/PCL.** (**/PCL** is a common extension for Pascal source programs.)

- A drive number. Example: **:1**
  Specifying no drive number means that your program will be stored on Drive Ø (the bottom drive of a floppy system or the Hard Disk in a Hard Disk system).

The complete filename might look like: **MYFILE/PCL:1**

Now, repeat the process, specifying O for object code, and using a different filename extension (for example, **MYFILE/OBJ:1**) to distinguish the object code from the source code.

5.e. Exiting Network Pascal

Once your program has been moved onto a disk at the Host system, you can safely exit Network Pascal.

At the Main Menu, type **N** to exit. As a safety precaution, the computer will ask you to confirm that you really want to exit. At the message **Are you sure?** type **Y** for "yes."

5.f. Using LINKLOAD to Run the Program

When the **Network 3** prompt is showing, you can run the program using the module **LINKLOAD**, included on the Network Pascal disk. At the **Network 3** prompt, type **LINKLOAD** and press **ENTER**. You'll see a row of options.

L=Load, R=Run, N=Network, I=Init, S=Symbols, B=Build CMD
>>

First, let's load the program. At the prompt (">>"), type **L** and press **ENTER**. The computer screen will prompt you for the name of the file (**"FILE="**). Type the name of the Pascal object program (for example, **MYFILE/OBJ:1**) and press **ENTER**. When the program is loaded, you'll see the name that was typed on the first line of the program (in this case, **TEST**) and the number of bytes of memory left.

Now let's run the program. At the >> prompt, type **R** and press **ENTER**. As each of the following prompts appears, press **ENTER** once more:

STACK SIZE:

INPUT =

OUTPUT =

(Pressing **ENTER** for **OUTPUT** simply sends the program's output -- the message it prints -- to the screen. **STACK SIZE** and **INPUT** aren't things we have to worry about for our example program.)

The program then runs. You should see the following message appear on the screen:

*** I am a Pascal wizard.**

When the **LINKLOAD** options reappear, you can type **N** and press **ENTER** to get back to the **Network 3** prompt.

CONGRATULATIONS! You have now successfully entered, compiled, and run a short Pascal program. Before we leave this demonstration, let's learn how to move Pascal programs back into student station memory. Let's begin by entering **NETPCL** at the **Network 3** prompt to get back into the Network Pascal editor/compiler.

5.g. Moving a Program into the Student Station's Memory

Remember that earlier in this demonstration you moved your program's source and object code from student station memory onto a disk at the Host system. It is also possible to move Pascal source and object code from the Host disk to the Student Station memory. At the Main Menu select **E**, and confirm that the editor's storage area is empty. Then return to the Main Menu by pressing **CLEAR**, then **C**, then typing **exit** and pressing **ENTER.**

Now let's move your program's source code from the Host disk. (You can also move the object code, but only the source code can be viewed using the editor.) At the Main Menu, select **M.** Next, select **S** for source code and **R** to read from disk. When asked for the filename, enter the filename that you used when you saved the source code.

Once the file is transferred, the Main Menu will reappear. Now select **E**, and note that the source code that was read from the Host's disk is listed in memory, ready to be edited.

Now let's return to the Main Menu and explore one last option, the **D** option to Delete file.

5.h. Deleting Files from Memory

The **D** option to Delete file may be used to erase the current source file and/or object file from memory, making room for another program's source and object code.

At this point in the demonstration, you have only source code in memory. (Since you have the source code saved on the Host's disk, you can delete it from memory and still retain the option of moving it back into memory at another time.)

Start by selecting **D** at the Main Menu. At the message **Source or Object (S/O)**: specify source or object (in this case, **S** for source). (Or, if you selected **D** by accident, press **<CLEAR>** instead of specifying source or object, to return to the Main Menu.)

Once source or object is specified, the file is erased and the Main Menu reappears.

You are now ready to begin the Pascal TUTORIAL. For more information about the Network Pascal editor/compiler, or for information on program development using Pascal for stand-alone systems, consult the EDITOR GUIDE (section C of this manual). This BEGINNER'S GUIDE ends with a summary of some special keys and commands available in the Network Pascal editor.

6. NETWORK EDITOR COMMAND SUMMARY

The following keys may be used to move the cursor, insert text, and delete text. See the Appendix to the EDITOR GUIDE for a complete list.

NOTE: When **SHIFT** and a key are specified below, press **SHIFT** while you press the key. When **CLEAR** and a key are specified, press **CLEAR** and then press the key.

Key	Function
↑	move cursor up
↓	move cursor down
←	move cursor left
→	move cursor right
ENTER	move cursor to beginning of next line
CLEAR ↑	scroll (move quickly) up
CLEAR ↓	scroll down
CLEAR ←	send cursor to beginning of current line
CLEAR →	send cursor to end of current line
SHIFT →	delete character
SHIFT ←	delete line
SHIFT @	insert line
CLEAR E	insert block of 16 lines
CLEAR ?	see how much memory remains
CLEAR C	get into the editor's "command mode"

Two command mode commands that may be useful to you at this point are shown below. The remaining commands are discussed in the EDITOR GUIDE.

Exit	exit editor (return to Main Menu)
Quit	exit editor and delete current source program

APPENDIX I: NETWORK CONFIGURATION INFORMATION

Several network configurations are possible. In making your choice, you should consider storage space and student access time:

Host Configurations

If students will be saving their programs on a disk at the Host system, be sure to choose a configuration that allows for maximum free disk space.

1. Floppy System

   The typical floppy-controlled Network system has the TRS-80 Network Operating System disk in Drive Ø. Diskettes containing programs to be run are then placed in Drive 1 and in any attached external drives. Network Pascal can be run in this manner -- with the Network Pascal diskette (and any other formatted disks, used to store Pascal programs) in Drive 1 and/or higher drives.

   To get more storage space, you can copy all programs on the Network Pascal disk onto your working copy of the Network Operating System, for use in Drive Ø. A formatted data diskette can then be placed in Drive 1, to store student programs.

2. Hard Disk System

   Programs on the Network Pascal diskette can be copied onto a Hard Disk and used without alteration. This provides more storage space and faster loading than is available with floppy-controlled networks. See your Hard Disk Network Operating System manual for general instructions on transferring files to the Hard Disk. (Be careful that the files you transfer do not have identical filenames to files already on the Hard Disk.)

   Programs on the two Model III Pascal diskettes (for a stand alone system) must be **PATCH**ed before being transferred to a Hard Disk. (You don't really need them on the Hard Disk, unless you plan to sometimes use the Host as a stand-alone system with the Hard Disk.) Information on performing this **PATCH** is in the MODEL III PASCAL SYSTEM REFERENCE GUIDE (section F).

Student Station Configurations

Network Pascal was designed to minimize disk access time. Once the Network Pascal editor/compiler program is loaded into a student station, the student can edit a program and then compile it with minimal impact on the network. The Pascal source code and object code for the student's program are stored in memory at the student station. During editing or compiling, the network is not used. In switching between editing and compiling, a small amount of time is required for loading program overlays.

Several configurations are possible. Option 1 is the best configuration when many students are using the Network 3 system.

1. The NETPCL editor/compiler (provided on the Network Pascal disk) is running in 3/4 of the stations, with LINKLOAD (also a program on the Network Pascal disk) set up as the "Run" module in the remaining 1/4. Students can edit, compile, and save their programs at the stations running NETPCL. Then they can move to the LINKLOAD stations to run programs, and avoid having to re-load LINKLOAD each time.

2. The editor/compiler is running in all of the stations, with several stand-alone Model III or 4 stations available to allow students to use the RUN program (provided on stand-alone system diskette) to run programs. (This involves removing from the Network the disk on which the student programs were saved, so that you can use that disk at the stand-alone system, OR copying student programs from the disk in the Network to a disk that will be used at the stand-alone system.)

3. Each student is assigned a station, where he or she loads the editor/compiler or the LINKLOAD or RUN program as needed (RUN is also provided on the Network Pascal diskette). This configuration may be appropriate when only a few students are using the network. If many students are using the network, disk access time required for this option may be prohibitive.

Network Pascal can handle small and medium-sized programs of up to 100 to 150 lines. This size of program is typical for teaching situations, particularly in beginning classes. The stand-alone disks in this package can be used at stand-alone Model III or 4 disk systems to allow for longer and more complex programs.

APPENDIX II: BACKUP INFORMATION

Before doing anything else, you should make a backup copy of the three diskettes provided with this package. These backup copies should be used as working diskettes, for everyday use. The original diskettes provided with the package should be stored in a safe place to protect them from damage.

To make backup copies, complete these steps at a two-drive stand-alone Model III or 4 disk system (the network's Host can be used as a stand-alone system for these steps).

MAKING A BACKUP COPY OF THE NETWORK PASCAL DISKETTE

1. Turn on the computer. (The on/off switch is located under the right edge of the keyboard.)

2. When the red light on the disk drive goes off, insert any Model III TRSDOS system diskette into Drive Ø (the bottom drive next to the video screen), with the square notch to the left and the label facing up. Then close the disk drive door.

   (A TRSDOS system diskette is any diskette that contains the TRSDOS operating system; for example, a TRSDOS diskette was provided when you purchased the Host computer.)

3. Insert a new, blank diskette into Drive 1 (the top drive next to the video screen). Then close the disk drive door.

4. Press the orange Reset button.

5. When you see the message **Enter Date MM/DD/YY**, type the date, using two digits each for the month, day, and year. (Example: **02/22/85** for February 22, 1985.) Then press **ENTER**.

6. When you see the message **Enter Time HH:MM:SS**, press **ENTER** to skip the time.

7. When you see **TRSDOS Ready**, type **BACKUP** and press **ENTER**.

8. When you see the message **SOURCE Drive Number?**, change diskettes as described below:

   a. Remove the TRSDOS system diskette from Drive Ø.

   b. Place an adhesive tab (provided with new diskettes) over the square notch on the Network Pascal diskette. (If you do not have any adhesive tabs, use a small piece of cellophane tape.)

   c. Insert the Network Pascal diskette into Drive Ø, with the covered square notch to the left and the label facing up. Then close the disk drive door.

9. Now, for **SOURCE Drive Number?**, type Ø and press **ENTER.**

10. For **DESTINATION Drive Number?**, type 1 and press **ENTER.**

11. For **SOURCE Disk Master Password?**, type **PASSWORD** and press **ENTER.**

12. The computer will proceed to make a backup copy of the Network Pascal diskette. When you see the message:

    ****Backup Complete****
    Insert SYSTEM Diskette <ENTER>

    remove the Network Pascal diskette from Drive Ø, and replace it in its protective envelope. Insert the TRSDOS system diskette into Drive Ø, close the disk drive door, and press **ENTER.** The diskette in Drive 1 is now an exact copy of the Network Pascal disk.

If, after the drives stop spinning, the message ****Backup Complete**** does not appear, or if you see an error message of any kind, then remove the Network Pascal disk from Drive Ø and replace it with the TRSDOS system diskette. Press the orange Reset button and go back to step 7. If an error still occurs, then get a new blank diskette or bulk erase the diskette you have been using as the destination disk. With the TRSDOS system diskette in Drive Ø and the new diskette in Drive 1, repeat the instructions from step 7.

MAKING A BACKUP COPY OF THE TWO MODEL III PASCAL DISKETTES

Follow the steps below for each diskette that you want to make a backup copy of.

1. Turn on the computer. (The on/off switch is located under the right edge of the keyboard.)

2. When the red light on the disk drive goes off, insert the Model III Pascal diskette into Drive Ø (the bottom drive next to the video screen), with the square notch to the left and the label facing up. Then close the disk drive door.

3. Insert a new, blank diskette into Drive 1 (the top drive next to the video screen). Then close the disk drive door.

4. Press the orange Reset button.

5. When you see the message **Enter Date MM/DD/YY**, type the date, using two digits each for the month, day, and year. (Example: Ø2/22/85 for February 22, 1985.) Then press **ENTER.**

6. When you see the message **Enter Time HH:MM:SS**, press **ENTER** to skip the time.

7. When you see **TRSDOS Ready**, type **BACKUP** and press **ENTER.**

8. For **SOURCE Drive Number?**, type Ø and press **ENTER.**

9. For **DESTINATION Drive Number?**, type 1 and press **ENTER.**

10. For **SOURCE Disk Master Password?**, type **PASSWORD** and press **ENTER.**

11. The computer will proceed to make a backup copy of the diskette in Drive Ø, onto the diskette in Drive 1. When the backup process is complete, you will see the message ****Backup Complete**** and **TRSDOS Ready** will reappear.

If, after the drives stop spinning, the message ****Backup Complete**** does not appear, or if you see an error message of any kind, press the orange Reset button and repeat the instructions from step 7. If an error still occurs, then get a new blank diskette or bulk erase the diskette you have been using as destination disk. Then insert the blank diskette in Drive 1, press the Reset button, and go to step 7.

SECTION B: TUTORIAL

TABLE OF CONTENTS

INTRODUCTION

Pascal was created by Professor Nicklaus Wirth at the Swiss Technical Institute in Zurich, Switzerland. It was first announced in 1965, when the languages most used by the computer industry were Fortran and COBOL, and the language most used for introducing students (in universities and elsewhere) to computer programming was Algol. Wirth thought languages like Fortran and COBOL were too loosely structured to promote good programming habits to students. Algol, although more structured, had significant drawbacks. And so, Wirth decided to depart from standard teaching practice and design a new language patterned after Algol. Pascal became his teaching language.

Since the first implementation of Pascal on the CDC-6600 computer system in 1971, Pascal has become one of the most popular programming languages in existence.

Pascal was created to make the development of computer programs a structured and logical process. Pascal contains the best features of most high-level programming languages, and for this reason Pascal is widely used in structured programming classes at the college level. Structured programming classes emphasize the use of guidelines and rules for developing computer programs. Some of the goals of structured programming are to encourage modularity and functionality, promote good documentation, and generate programs that have smooth flows of logic from beginning to end.

Features of Pascal include the following:

- The powerful ability to build new data types and structures as desired.
- The control statements **WHILE**, **REPEAT**, **FOR**, **IF**, **CASE**, **GOTO**.
- The logical operators **AND**, **OR**, **NOT**.
- The relational operators: equal to, less than, greater than, less than or equal to, greater than or equal to, not equal to.
- Recursive procedures and functions with parameter lists.
- The ability to insert blanks and comments into the source program easily. The ability to use long variable names, with no space or time penalty.
- User-controlled dynamic memory management.
- Efficient memory management of variables, functions and procedures.
- Arrays of one or more dimension.
- Record data structures.

- Sets and set operations.

- Subrange and enumerated data types.

- Named constants.

- Read and write statements plus formatted write statements.

- Built-in functions and procedures.

Network Pascal and Model III Pascal (the stand-alone system version) are full implementations of the standard Pascal language. Thus, program portability is greatly enhanced. Programs generated by TRS-80 Pascal execute much faster than programs generated by most BASIC systems, making TRS-80 Pascal a logical choice as a general high-level programming language.

CHAPTER ONE

INTRODUCTION TO PASCAL STATEMENTS

PROGRAM STRUCTURE

A Pascal program is largely composed of statements. A statement is a Pascal language word that instructs the computer to perform a specific action. A statement may have arguments that provide information needed to perform the action. For example: a statement that tells the computer to write a message also should tell the computer what message to write and where to write it.

Chapter One introduces some basic Pascal statements. Later chapters will introduce more advanced statements, and will introduce the other kinds of information needed in a Pascal program.

Three basic statements form the simplest level in a Pascal program: **PROGRAM**, **BEGIN**, and **END**. These words can be thought of as the "outer shell" that must be around all programs. Listing 1.1 is a legal Pascal program. However, it does nothing because it contains no additional statements. It simply begins, and then ends:

Listing 1.1

```
PROGRAM test;
BEGIN
END.
```

PROGRAM marks the beginning of a Pascal program. As you see in Listing 1.1, this statement also provides a place for the programmer to give the program a name. A semicolon (;) must follow this statement.

BEGIN marks the beginning of the active part of the program. Statements to make the computer process information are placed after the **BEGIN** statement. (Chapter Two will introduce you to information that may lie before **BEGIN**.)

END followed by a period (.) marks the end of the entire program. By adding a **WRITELN** statement (think "write-line") to the listing of 1.1, we can make the program do something:

Listing 1.2

```
PROGRAM test;
BEGIN
   WRITELN(OUTPUT,'* Pascal is a very structured language.');
   WRITELN(OUTPUT,'* It promotes good programming habits.');
END.
```

If someone ran the program of Listing 1.2, the program would write the following message:

*** Pascal is a very structured language.**
*** It promotes good programming habits.**

The argument **OUTPUT** in Listing 1.2 designates where the message is written. **OUTPUT** is a buffer, or holding area within the program. When the program is run, the user can indicate whether **OUTPUT** represents a specific disk file, a printer, or the video screen. The message will then be routed to the correct destination. This kind of buffer is known in Pascal as a "file," but this "logical file" should not be confused with a physical file for permanent storage on diskette.

PROGRAM STYLE

Let's take a closer look at Listing 1.2, so that we can begin to get a feel for Pascal programming style.

Notice that **BEGIN** and **END** are the only two lines in the program that do not end with a semicolon (;). Semicolons are required after most Pascal programming statements. For now, a good rule of thumb is to always include a semicolon after a Pascal statement.

> NOTE: A Pascal statement that immediately precedes an **END** statement is not required to end with a semicolon, but the programmer can place a semicolon there if he or she desires. Program listings in this manual illustrate both options. This rule is true whether the particular **END** statement marks the end of the entire program, or just the end of a sub-program (discussed later).

Notice how the program of Listing 1.2 is formatted. Lines in a Pascal program may be indented at the programmer's discretion for easy reading. Spaces may also be inserted for easy reading. One restriction is that a message to be printed by a single **WRITELN** statement has to appear all on one line. (In Listing 1.2, two **WRITELN** statements had to be used for this very reason.) The message to be printed is enclosed in single quotes. These messages are called "text strings," and may include letters, numbers, or most other characters.

The program name in Listing 1.2 is "**test**", but any name could have been used, as long as the first character was a letter. Upper or lower case may be used. Spaces cannot appear in the program name.

THE WRITE AND WRITELN STATEMENTS

Let's look at another example. Listing 1.3 introduces the **WRITE** statement. **WRITE** is similar to **WRITELN** with one important difference. **WRITELN** prints a message and then executes a carriage return, causing the next message to be printed on a new line. **WRITE** prints consecutive messages on the same line, until a line is filled, and only then is a new line started.

Listing 1.3

```
PROGRAM test;
   BEGIN
   (* the purpose of this program is to give an example *)
   (* of how to use the WRITE and WRITELN statements    *)
   WRITE(OUTPUT,' * Now is the time');
   WRITE(OUTPUT,' for all good programmers');
   WRITE(OUTPUT,' to learn');
   WRITELN(OUTPUT,' Pascal.');
   (* The next message starts on a new line *)
   WRITELN(OUTPUT,' * You will become a Pascal magician.');
   END.
```

If someone ran the program of Listing 1.3, the following message would be written to "**OUTPUT**":

```
* Now is the time for all good programmers to learn Pascal.
* You will become a Pascal magician.
```

PROGRAMMER COMMENTS

Listing 1.3 also demonstrates programmer comments. Notice the third and fourth lines of the program, and the third line from the bottom. These lines contain comments enclosed in parentheses and asterisks. Unlike the messages in the **WRITE** statements, these comments are not written to a file.

Comments are distinguished from program statements by being enclosed in a pair of parentheses and a pair of asterisks:

```
(* This is an example of a comment.  Testing, 1 2 3.*)
```

Braces may optionally replace parentheses and asterisks. When using Pascal on a Model III or 4, you may create the left brace ({) by pressing **CLEAR** and then **4**. **CLEAR** and then **5** will create the right brace (}).

Comments, inserted into Pascal programs in the format shown, allow the programmer to document what his or her program does. Later, when the programmer or another person is reading through the program, the comments aid the reader. Use of comments is particularly important in long, complex programs.

Note that the asterisks in the **WRITE** and **WRITELN** statements are printed along with the message. Any characters typed within the single quotes in a message are considered to be part of the message.

Now that you have learned some of the basics of Pascal program structure, let's take a short quiz and then complete a hands-on activity.

CHAPTER ONE QUIZ

1. The first statement of a Pascal program is the ________________ statement.

2. The ________________ statement will cause the next message to be written on a new line.

3. The ________________ statement will allow the next message that is written to begin on the same line, if there is room on the line.

4. Most Pascal statements are followed by a(n) ________________.

5. The ________________ statement is the last statement of a program.

CHAPTER ONE ACTIVITY

Complete the following activities at a Network 3 Student Station. If you need more help with entering the program than is given below, take a look at the Summary of Editor Commands in the Appendix to the EDITOR GUIDE (page C-15) or on page A-8 of the BEGINNER'S GUIDE.

1. When the **Network 3** prompt is showing, type **NETPCL** and press **ENTER** to get into Network Pascal.

2. A menu of options will appear. When the menu is showing, press **E** to use the Network Pascal editor.

3. Enter the first line of the program of Listing 1.3 by completing these steps:

   a. Insert a blank line by holding down **SHIFT** and pressing **@** once.

   b. Type the line (in this case **PROGRAM test;**), then press **ENTER.** (To switch between lowercase and uppercase mode, press **SHIFT Ø**, or get into lowercase mode and then hold down **SHIFT** each time you want to type a capital letter.)

4. Enter all the remaining lines of Listing 1.3. To indent, press the space bar a few times at the beginning of the line. To correct a typing error, backspace using the left-arrow key.

   For now, make sure that you type each line exactly as shown.

5. When the entire program is entered, press **CLEAR** and then **C.** When the angle brackets appear (<>), type **EXIT** and press **ENTER.** This will get you back to the menu.

6. At the menu, type **P** to load the Pascal compiler. (Loading the compiler may take a few moments.)

7. When the compiler is loaded, you'll see the question "**Generate object? (Y/N)**". Type **Y.**

8. When you see the message "**Print listing on printer?**", type **N.**

9. The program will be compiled, and the compiler will tell you how many errors were detected.

   a. If you typed the program exactly as it appears in this book, no errors were detected. Press **ENTER** to exit the compiler and then go to step 10.

   b. If the compiler detected any errors, press **ENTER** and then type **E** to use the editor again. Proofread the program carefully for errors. The most common errors will be in punctuation, so make sure that your semicolons, parentheses, commas, quote marks, and asterisks are in place. Correct any errors, and then repeat these steps from step 5 above. (One typing mistake can cause many errors, so if you only see one error, try compiling again after you have corrected that error.)

10. At the menu, type **M** to move the program onto a diskette at the host system. At the message "**Source or Object (S/O):**" type **O** to copy the object (executable) program onto diskette.

    At the message "**Read from or Write to disk:**" type **W.**

    At the message "**File name:**" type in a unique name -- a name that is different from any name that anyone else in the class is using. Then type the extension **/OBJ.** Finally, press **ENTER.** For example:

    GEORGE1/OBJ

11. The menu will reappear when your file has been properly copied. NOW you can exit to Network 3 by typing **N** when the menu is displayed.

12. At the **Network 3** prompt, type **LINKLOAD** and press **ENTER.**

13. When the LINKLOAD options appear, type **L** and press **ENTER.** When prompted for the file, enter the name that you entered at step 10.

14. When the angle brackets reappear, enter **R** to run the program. When each of the following three prompts appear, press **ENTER:**

    STACK SIZE:
    INPUT =
    OUTPUT =

If you have followed the above steps correctly, you'll see this message printed on the screen:

*** Now is the time for all good programmers to learn Pascal.**
*** You will become a Pascal magician.**

When you are ready and the angle brackets are showing, type **N** and press **ENTER** to return to the **Network 3** prompt.

CHAPTER TWO

USING VARIABLES AND CONSTANTS TO HANDLE DATA

Chapter One introduced some of the statements found in a Pascal program. Chapter Two continues by introducing variables and constants, which are used to hold information that the programmer wishes to manipulate.

VARIABLES

Variables in Pascal serve the same purpose as they do in most other programming languages. That is, they can be used within a program to temporarily store information (numbers, letters, and special characters) that the programmer wishes to manipulate. A single Pascal variable can hold different values at different points in a program. However, a single variable can only hold one kind, or type, of information (an alphanumeric character, an integer, etc.).

Think of a variable as a box that can hold a certain value. At any point in the program, the programmer can put a value into the box, or can take a value out of the box to put it elsewhere. The program can be made to check the value currently in the box, and can perform certain actions based on that value. A single variable can only hold one value at a time.

VARIABLE NAMES

Each variable must be given a name by the programmer. The name identifies the particular variable. Each variable name must start with a letter. It may be composed of as many letters and digits as desired, in any combination. Good programming practice dictates that the name reflect the kind of value to be held. For example: **employeenumber**, **price**, **testgrade.** This is not required by Pascal, but does make the program easier to read and understand.

"Reserved words" must not be used as variable names. A reserved word in Pascal is a keyword that has a set meaning, no matter which Pascal program it is used in. For example, **BEGIN** and **END**, introduced in Chapter One, are reserved words. A complete list of these words can be found on page D-5.

> NOTE: Alcor Pascal requires that the first eight characters of a variable name be a unique name within the program.

VARIABLE TYPES

In order to use a variable, you must first declare it. That is, you must formally associate the variable with the type of information that it will hold. A variable named "**taxnumber**" that might take on the value of 1 to 100 at any time in the program is an example of the variable type **INTEGER**. (The value of "**taxnumber**" will always be expressed as an integer.)

The simple variable types that are predefined by Pascal are outlined below. They are: **INTEGER**, **REAL**, **CHAR**, **TEXT**, and **BOOLEAN**. (Other types are available, but they will be discussed in later chapters.)

DECLARING VARIABLES

Declaring a variable is done in the "**VAR**" section of the program. The **VAR** section consists of the word **VAR** followed by any number of variable declarations. This section is placed between the **PROGRAM** statement and the first **BEGIN** statement of the program.* For example:

Listing 2.1

```
PROGRAM test;
VAR
    taxnumber:INTEGER;
BEGIN
END.
```

The **VAR** section of Listing 2.1 declares the variable **taxnumber** to be of the type **INTEGER**. A variable declaration has the form:

```
variablename:TYPE;
```

A colon (:) separates the variable name from the variable type, and a semicolon (;) must follow each variable declaration.

The declared section may be indented for easy reading, if the programmer desires.

Several variables of the same type may be declared on a single line. For example:

```
taxnumber,employeenumber,deptnumber,jobnumber:INTEGER;
```

* Programs with more than one **BEGIN** statement are covered in later chapters.

INTEGER VARIABLES

Variables that will be used to store whole numbers are of the type **INTEGER**. The maximum and minimum size of the whole numbers that can be stored depends on the computer. On most microcomputers, including the TRS-80 Model III, the range is -32768 to +32767.

REAL VARIABLES

Variables that will be used to store numbers with fractional or decimal values are of the type **REAL**. (The number 2.98 is an example of a real number.) Real numbers must start with a digit and may contain a decimal point. They do not have to contain a decimal point. If a decimal point is present, it must be followed by a digit. Example: The numbers .009 and 10. are illegal by these rules, while 0.009 and 10 or 10.00 (or 10.0, 10.000, etc.) are legal ways to express the same numbers.

A variable of the type **REAL** might represent a dollar selling price or a checkbook entry. It might be declared as follows:

Listing 2.2

```
PROGRAM test;
VAR
    taxnumber:INTEGER;
    cost:REAL;
BEGIN
END.
```

CHAR VARIABLES

Variables that will be used to store single characters are of the type **CHAR**. The character assigned to the variable may be a letter, digit, or special symbol such as % , enclosed in single quotatation marks. A **CHAR** variable represents only one character at a time.

TEXT VARIABLES

A variable that will be used to store or retrieve information using files is of type **TEXT**. In Chapter One, you learned how a string of characters in single quotation marks could be written to a file called **OUTPUT**. This writing process can be compared to the way that a value is assigned to a variable. In fact, variables of type **TEXT** are always used in output and input. That is, they serve as variables for routing messages between the program and disk files, the screen, or a printer.

A variable of type **TEXT** can hold a string of one of more characters.

BOOLEAN VARIABLES

A variable declared to be of the type **BOOLEAN** may store one of two values: **TRUE** or **FALSE**. Boolean variables are primarily used in flow-control statements -- statements that can make the program follow different paths of execution depending on different conditions. **BOOLEAN** variables are typically used with the control statements **WHILE**, **IF** and **REPEAT**, covered in later chapters.

CONSTANTS

While a variable can hold different values at different points in the program, a constant has a fixed value. In Pascal, constants are declared in the **CONST** section of the program, which is placed between the program name and the first **BEGIN** statement of the program. Constants are declared using the format:

```
        name = constant;
```

(Spacing between the name, equal sign, and constant is flexible.) For example:

Listing 2.3

```
PROGRAM test;
CONST
    pi     = 3.141597;
    maxnum = 2000;
    tstring= ' I am a Pascal wizard.';
VAR
    taxnumber:INTEGER;
BEGIN
END.
```

As shown in Listing 2.3, constants may be real numbers, integers, or text strings. (A text-string constant is any string of characters enclosed by single quotes.)

CHAPTER TWO QUIZZ

1. VARIABLES serve as storage areas for information that the programmer may wish to manipulate.

2. Variables are declared in the VAR section of the program.

3. Five predefined types of variables in Pascal are CHAR, INTERGER, TEXT, BOOLEAN, and REAL.

4. The syntax of a variable declaration is :
   VAR
   variablename: TYPE ;

5. Variables declared as the type VALUE may take on the value of a letter, a digit, or a special symbol.

6. A variable declared to be of the type TEXT may be used to route information between the program and various input and output devices.

7. A value that is fixed in the program (it will not change) may be declared as a CONSTANT in the CONST section of the program.

CHAPTER TWO ACTIVITY

For each of the values below, name a variable type discussed in Chapter Two that could include that value. Challenge question: Could any of these variables be matched with more than one type?

	VALUE	VARIABLE TYPE
1.	**1500.25**	REAL
2.	**1500**	INTERGER
3.	**%**	CHAR
4.	**I am a Pascal wizard.**	TEXT

CHAPTER THREE

INPUT AND OUTPUT -- AN INTRODUCTION

Can you guess what the program of Listing 3.1 would do if you ran it?

Listing 3.1

```
PROGRAM alpha;
CONST
   pi      = 3.141597;
   maxtax  = 2000;
   tstring = ' I am a Pascal Wizard';
VAR
   out    :TEXT;
   max    :REAL;
   number:INTEGER;
BEGIN
   REWRITE(out);
   WRITELN(OUTPUT,'Program starting execution.');
   WRITELN(' The value pi = ',pi);
   WRITELN(' The value maxtax = ',maxtax);
   WRITELN(tstring);
   WRITELN(out,'This program tests file I/O');
   WRITELN(OUTPUT,'Program finished.');
END.
```

The program would write messages to the files **OUTPUT** and **out**. Because **OUTPUT** and **out** both receive information, they are known as "output" files -- the program "outputs" information to them.

But in Listing 3.1, there are some extra steps that you didn't see when **WRITE** and **WRITELN** were used in Chapter One. These extra steps are required because this time two different output files are used. The file "**out**" had to be specially created and initialized before it could be used.

WRITING TO FILES OTHER THAN "OUTPUT"

In Pascal, the file **OUTPUT** enjoys a special status. To write to a file of any other name, you must complete certain steps that are not required for **OUTPUT**. Let's take a look at these steps.

1. Declare the file as a **TEXT** variable in the **VAR** section of the program.

   Notice that **out** is declared in the **VAR** section of Listing 3.1 as a **TEXT** variable. Remember from Chapter Two that the **TEXT** variable can act as an "extra" file for routing information in and out of the program. Whenever more than one file is needed in a Pascal program, a file not named **OUTPUT** must be created as a **TEXT** variable.

2. Use the **REWRITE** statement to initialize the file.

   REWRITE must be used (only once in the program) for each file other than **OUTPUT** that you are writing to. The **REWRITE** statement opens the file that is named in its argument, and makes the file ready to receive information. Then **WRITELN** or **WRITE** can be used on the file. Any information previously in the file is lost when **REWRITE** is used.

   The form of the **REWRITE** statement is:

   ```
   REWRITE(variablename);
   ```

3. Provide two arguments for the **WRITELN** or **WRITE** statement.

   The **WRITELN** or **WRITE** statement normally has two arguments. The first argument names what is written to (for example: **OUTPUT**). The second argument names what is written (for example, **' This message.'**).

   Notice that the second, third, and fourth **WRITELN** statements of Listing 3.1 have only one argument -- the message. If you are writing to the file named **OUTPUT**, Pascal does not require the file to be named. If the file is not named, Pascal assumes it is **OUTPUT**.

   Writing to any file other than **OUTPUT** requires that the name be mentioned.

INPUT: READING INFORMATION FROM A FILE

Files that the program reads information from are called "input" files. Information is "input" into the program using these files. Information is read from the input device (keyboard or disk file) that is pointed to by the input file.

The input file **INPUT** is similar to the output file **OUTPUT**. **INPUT** is the assumed filename for an input file. Extra steps are required when you read from any file not named **INPUT**. These steps are:

1. Declare the file as a **TEXT** variable in the **VAR** section of the program. For example:

   ```
   VAR
     in:TEXT;
   ```

2. Use the **RESET** statement to initialize the file.

   RESET is to input files as **REWRITE** is to output files. **RESET** prepares a file to have its information read into the program. **INPUT** is the only file that does not require use of **RESET** before it can be read. The format of the **RESET** statement is:

   ```
   RESET(variablename);
   ```

3. Provide two arguments for the **READ** and **READLN** statements. These statements are described below.

READ AND READLN: STATEMENTS USED FOR INPUT

The **READ** and **READLN** statements are the input equivalents of the **WRITE** and **WRITELN** statements. **READ** and **READLN** will read a value into the program.

Successive **READs** from the same file will cause a series of inputs from the same line of that file. These "lines" might be rows of characters in a disk file, or they might be lines that have been typed in at the keyboard and are displayed on the video screen.

When a **READ** is performed on an integer or real quantity in a text file, the **READ** statement scans the current line of the file until a non-blank character is found. Then it reads non-blank characters from that line until the conditions of the **READ** are satisfied. The next **READ** will begin scanning on the same line, right after the last character read, and so on, until the end of the line is reached. If the end of the line is reached before any integer is found, the scan will continue at the beginning of the next line.

The **READLN** statement performs the same function as **READ**, except that the cursor (acting as a place-marker) will always be moved to the beginning of the next line after all inputs to the statement are satisfied, even if the end of the line has not been reached.

When **INPUT** is the file being read, the **READLN** statement is not required to have arguments. The **READLN** statement without arguments waits for the computer operator to press **ENTER**, and then positions the cursor at the beginning of the next line without reading any values. (It can be used to skip a line or portion of a line.)

The arguments allowed for the **READ** statement are files and variables. (Text or numbers are read from the file into the variable.) If a **READ** statement does not have a file named as argument, the file is assumed to be **INPUT**. The following are some examples of legal **READ** and **READLN** statements:

```
READ(INPUT,taxnumber);
READ(taxnumber);
READ(in,taxnumber);
READLN;
READLN(INPUT,taxnumber);
READLN(taxnumber);
READLN(in,taxnumber);
```

INPUT -- AN EXAMPLE

An example may help illustrate the difference between **READ** and **READLN**, and may help familiarize you with input concepts. Suppose that you typed these lines in at the keyboard and they are displayed on the video display in this format:

```
ABCDEFG
HIJKLMN
OPQRSTU
VWXYZ
```

Now, suppose that your program is going to read part of this keyboard input into a variable of type **CHAR**. Assume that a variable named **letter** has been declared as type **CHAR**. The **READ** statement might look like this:

```
READ(INPUT,letter);
```

The first **READ(INPUT,letter);** statement of course reads the letter **A**. The cursor (as place marker) is then positioned at **B**, and the next **READ** or **READLN** statement reads beginning at **B**.

If we started with a **READLN** statement:

```
READLN(INPUT,letter);
```

the letter **A** would be read first. But, as soon as the **A** was read, **READLN** would make the cursor jump to the beginning of the next line. The next **READ** or **READLN** statement would read beginning with **H**.

CHAPTER THREE QUIZ

1. The predefined file variables ______________ and ____________ are not required to be declared in the **VAR** section as the type **TEXT.**

2. The first argument in the ______________ , ____________, ___________, and _____________ statements directs I/O to or from a file or device.

3. The purpose of the ______________ statement is to open a file and ready it for writing.

4. The purpose of the _____________ statement is to open a file and ready it for reading.

5. After execution of a(n) ______________ , the previous contents of an output file are lost.

6. A(n) _____________ or _____________ statement will cause a line feed immediately after input or output occurs.

CHAPTER THREE ACTIVITY

Try running the program on the following page. It will give you a chance to practice program I/O. If you need to, review the hands-on guidelines in the Chapter One Activity for editing, compiling, and running the program on the Network 3. When you run the program, press **ENTER** for **INPUT** and **OUTPUT** to direct messages to the video screen and to read messages from the keyboard.

NOTES:

- Open brackets ([) are produced by **CLEAR 1.** Close brackets are produced using **CLEAR 2.**

- Don't worry about the **PACKED ARRAY** declared in the **VAR** section. Packed arrays are introduced in Chapter Eight.

- "**tax:8:2**" in the third **WRITELN** statement from the end of the program illustrates an option available when you are writing real numbers to an output file. The first number specifies the total width of the field, while the second number specifies the number of digits after the decimal point.

Listing 3.2

```
PROGRAM testIO;
(* Purpose- the purpose of this program is to        *)
(* demonstrate I/O to a text file using integer and   *)
(* real input variables.                              *)
VAR
   taxnumbr,emnumber     :INTEGER;
   tax                   :REAL;
   ID                    :PACKED ARRAY[1..72]OF CHAR;
BEGIN
   WRITELN(OUTPUT,'* Enter your federal tax number: ');
   READLN(INPUT,taxnumbr);
   WRITELN(OUTPUT,'* Enter your dollar tax total: ');
   READLN(INPUT,tax);
   WRITELN
         (OUTPUT,' * Enter your employee number, a space,');
   WRITELN(OUTPUT,' followed by your business ID number:');
   READ(INPUT,emnumber);
   READLN(INPUT,ID);
   WRITELN(OUTPUT,' Tax number        = ',taxnumbr);
   WRITELN(OUTPUT,' Dollar tax total = ',tax:8:2);
   WRITELN(OUTPUT,' Employee number   = ',emnumber);
   WRITELN(OUTPUT,' Business I.D.     = ',ID);
END.
```

The following I/O will appear on the screen:

```
* Enter your federal tax number:
32000                         <user input>
* Enter your tax total:
2345.98                       <user input>
* Enter your employee number,a space,
followed by your business ID number:
23455 4669                    <user input>
Tax number        = 32000
Dollar tax total  = 2345.98
Employee number   = 23455
Business I.D.     = 4669
```

CHAPTER FOUR

ASSIGNING VALUES TO VARIABLES AND USING ARITHMETIC EXPRESSIONS

In previous chapters you learned how to declare variables, and you learned something about using varibles in input and output.

Now you will learn how to alter a variable's value by directly assigning a specific value to it.

USING ASSIGNMENT STATEMENTS

An assignment statement sets a variable's value equal to an expression. The expression may be the name of another variable, or it may be a series of arithmetic or boolean operations. The form used is as follows:

variablename1:=variablename2;

or

variablename:=expression;

if desired, spaces may be inserted between units of the statement for easier reading. For example: **variablename := expression;**

Statements of this form cause the variable on the left-hand side to take on the value of the variable or expression on the right-hand side. (The value on the right-hand side is unchanged.)

Listing 4.1 demonstrates the use of statements that assign values to a variable.

First, let's look at what the program of Listing 4.1 is doing. Let's assume that the input and output device used by this program is the video display. When the message "**Enter annual interest rate:**" is printed on the video display, the user enters an interest rate. This interest rate is read into the program by the line **READLN(intrate)** and is stored in the variable **intrate**, defined in the **VAR** section. Similarly, the variable **principle** is made to store the real number entered as principle.

Listing 4.1

```
Program MAGIC;
VAR
   intrate,principle,anint,calc:REAL;
BEGIN
   WRITELN(' ******* Interest rate problem *******');
   WRITELN(' Enter annual interest rate:');
   READLN(intrate);
   WRITELN(' Enter the principle amount of loan:');
   READLN(principle);
   calc:=intrate * principle;
   anint:=calc;
   WRITELN(' Your annual interest payment = ',anint);
END.
```

Finally, the assignment statement is used:

```
calc:=intrate * principle;
```

This statement employs the arithmetic operation **intrate * principle**, which multiplies the value of the variable **intrate** by the value of the variable **principle**. ("*" is the computer's symbol for multiplication.) The result of multiplication is assigned to the variable **calc**.

A second example of assignment in Listing 4.1 is the statement **anint:=calc**. This assignment simply passes the value of one variable to another variable. In this example, the value is passed only to be printed (from **anint**). The programmer might need to pass a value in this manner, for example, if a later part of the program used the value of **calc** for one purpose and the value of **anint** for another purpose.

Arithmetic Operators

In Listing 4.1 you saw the multiplication operator "*" used. Pascal includes seven arithmetic operators, shown in the table below. They are evaluated in order of the precedence shown below, with "1" first and "3" last. Operators of equal precedence are evaluated from left to right. Parentheses may also alter the order of evaluation.

Table 4.1

Symbol	Description	Precedence	Example
-	negates one number	1	-25
*	multiplies numbers	2	5 * 5 = 25
/	divides real numbers	2	25 / 2 = 12.5
div	divides integers	2	25 DIV 2 = 12
mod	divides integers and keeps the remainder as the result	2	25 MOD 2 = 1
+	adds	3	5 + 5 = 10
-	subtracts	3	5 - 5 = 0

Parentheses may alter the natural order of precedence. For example:

```
5 + 25  / 5 = 10

(5 + 25) / 5 = 6
```

Listing 4.2 illustrates the use of the arithmetic operators and parentheses.

Listing 4.2

```
PROGRAM math;
CONST
   fudge    = 100;
   lossacre = 0.50;
VAR
   acsoy,acgreen    :INTEGER;
   prsoy,prgreen    :REAL;
   profit,overcost :REAL;
BEGIN
   WRITELN(OUTPUT,' **** Farmer profit analysis program **** ');
   WRITELN(OUTPUT,'* Please enter the following information:');
   WRITELN(OUTPUT,'* Acres planted in soy beans = ');
   READLN(INPUT,acsoy);
   WRITELN(OUTPUT,'* Profit per acre of soybeans = ');
   READLN(INPUT,prsoy);
   WRITELN(OUTPUT,'* Acres planted in green beans = ');
   READLN(INPUT,acgreen);
   WRITELN(OUTPUT,'* Profit per acre of green beans = ');
   READLN(INPUT,prgreen);
   WRITELN(OUTPUT,'$$$ COMPUTATION IN PROGRESS $$$');
   profit:= acsoy * prsoy + acgreen * prgreen
            - (fudge / (acsoy+acgreen) * lossacre);
   WRITELN(OUTPUT, ' Your computed profit is ');
   WRITELN(OUTPUT,profit:10:2);
END.
```

The profit calculation (in the third line up from the **END** statement) uses parentheses to alter the normal operator precedence. If the normal precedence was followed, the calculation would yield the wrong result.

The order of evaluation without parentheses would be:

1. **acsoy** and **prsoy** multiplied
2. **acgreen** and **prgreen** multiplied
3. **fudge** divided by **acsoy**
4. **acgreen * lossacre**
5. **result1 + result2**
6. **result5 - result3**
7. **result4 + result6**

The desired result is obtained by including parentheses. The correct order of evaluation is:

```
   profit := acsoy * prsoy + acgreen * prgreen
             -(fudge / (acsoy+acgreen) * lossacre);
```

1. **acsoy** added to **acgreen**
2. **fudge / result1**
3. **result2 * lossacre**
4. **acsoy * prsoy**
5. **acgreen * prgreen**
6. **result4 + result5**
7. **result6 - result3**

CONSIDERATIONS FOR VARIABLES DESIGNED TO HOLD THE RESULTS OF ARITHMETIC OPERATIONS

If two numbers are operated on, the normal result will have a type that is dependent on the types of the two numbers. The variable types required to store the results of specific operations are summarized in the following table.

Table 4.2

*	multiply	real * integer = real result. integer * real = real result. real * real = real result. integer * integer = integer result.
/	real divide	real / real = real result. real/integer = real result. integer/real = real result. integer/integer = real result.
div	integer divide	integer div integer = integer result. integer arguments only.
mod		integer mod integer = integer (integer div integer= remainder)
+	add	integer + integer = integer result. integer + real = real result. real + integer = real result. real + real = real result.
-	subtract	integer - integer = integer result. integer - real = real result. real - integer = real result. real - real = real result.

CHAPTER FOUR QUIZ

1. ____________ is used to assign a value to a variable.

2. Operator precedence refers to the order in which an ________________ is evaluated.

3. The natural order of expression evaluation may be altered by using ___________________ .

4. The operator with the highest __________________ will be evaluated first.

5. Operators that have the same level of precedence will be evaluated in ______________ to _______________ order.

6. After execution of the following Pascal program, variable x will have the value _________________.

```
PROGRAM QUIZ;
VAR
  x:integer;
BEGIN
  x:=4 + 5 * 2;
END.
```

CHAPTER FOUR ACTIVITY

Fill in the missing parts of the program below. Then, run the program to test your work.

```
__________ average;
   VAR
   numberofwidgets          : _______________ ;
   ___________,costperwidget : REAL;
BEGIN
   WRITELN(OUTPUT,'This program calculates cost per widget.');
   WRITELN(OUTPUT,'How many widgets did you buy?');
   READLN(INPUT, ____________________);
   WRITELN(OUTPUT,'What was the total cost of the widgets?');
   READLN(INPUT,totalcost);
   _______________ := totalcost ___ numberofwidgets;
   WRITELN(OUTPUT,'The cost per widget = ', ________________);
END.
```

CHAPTER FIVE

CONTROLLING THE PROGRAM'S PATH OF EXECUTION

Statements in a Pascal program are not always executed in top-to-bottom order. The programmer can include statements in the program that alter the path of execution. A group of statements might be executed repeatedly before execution moves on to a second group of statements. Or, the programmer might include statements to make the computer execute one statement under certain conditions, but another statement under certain other conditions. Statements that are used to alter program flow in this way are called flow-control statements.

USING THE FOR STATEMENT TO LOOP

Executing a statement or series of statements a predetermined number of times is called looping. If you wish to loop in a Pascal program, you should use the **FOR** statement. Let's look at the following example .

Listing 5.1

```
PROGRAM math;
CONST
    fudge    = 100;
    lossacre = 0.50;
    prsoy    = 195.98;
    prgreen  = 200.56;
VAR
    acsoy,acgreen,nofields,select,fieldnumber:INTEGER;
    profit,overcost:REAL;
BEGIN
    WRITELN(OUTPUT,'* Farmer planting analysis program * ');
    WRITELN(OUTPUT,'* How many fields do you have?');
    READLN(INPUT,nofields);
    FOR fieldnumber := 1 to nofields DO
        BEGIN
        WRITELN(OUTPUT,'* For field number',fieldnumber);
        WRITELN(OUTPUT,'* Acres planted in soy beans = ');
        READLN(INPUT,acsoy);
        WRITELN(OUTPUT,'* Acres planted in green beans = ');
        READLN(INPUT,acgreen);
        profit:= acsoy * prsoy + acgreen * prgreen
                      - (fudge / (acsoy+acgreen) * lossacre);
        WRITE(OUTPUT, '* Your computed profit for field number  '
                      ,fieldnumber,'   is ');
        WRITELN(OUTPUT,profit:12:2);
        END;
END.
```

Notice that the program of Listing 5.1 contains two **BEGIN** and two **END** statements. The first **END** statement is not followed by a period (marking the end of the program) but instead is followed, like most program lines, by a semicolon. This is our first example of a compound statement -- a sub-program within the program.

A series of program statements surrounded by a **BEGIN** and an **END** statement is considered a compound statement. Compound statements are executed as a set, as if they formed a single statement. In Pascal, anywhere a single statement may be used, a compound statement may be used.

In Listing 5.1, the variable **fieldnumber** controls the repeated execution of the compound statement. **Fieldnumber** is declared as an integer. **"For fieldnumber :=1 to nofields"** gives **fieldnumber** the value of one as the computer begins to execute the compound statement for the first time. (The **DO** on the end of the **FOR** statement tells the computer to "do" the compound statement until the value of **fieldnumber** becomes equal to the value of **nofields** -- the total number of fields owned by the farmer.)

Each time the compound statement is repeated, the value of **fieldnumber** is incremented by one, until its value is greater than **nofields.** At that point, the loop stops, and control is passed to the next statement in the program.

The lower and upper limits controlling the loop may be variables (like **nofields**), constants (like **1**), or arithmetic expressions. The expression is evaluated each time the computer is about to begin the loop. The upper bound must be greater than or equal to the lower bound for the loop to execute at least once.

A variation on the **FOR** loop just described causes the loop control variable to be decremented by one instead of incremented by one. The syntax for this is the same as above except that the **to** in the **FOR** statement is replaced with **downto.** The initial upper bound on the loop control variable must be larger than or equal to the lower bound for the loop to execute at least once.

Examples of legal FOR statements

```
FOR a:=1 to 10 DO
FOR a:=1 to b DO
FOR a:=b - 1 to d * 3 DO
FOR a:=b downto 1 DO
FOR a:=100 downto b DO
FOR a:=b * 3 downto d - 1 DO
```

USING THE CASE STATEMENT

The CASE statement is used as a selection-control statement. That is, it is used when you need to select for execution one statement from a list of statements.

Let's take a look at Listing 5.2. In front of every statement in the list that follows "**CASE**" is a case-selector constant. This selector value must be of the same type as the case-selector variable, and may be composed of a list of values for each statement it precedes. "**END;**" (with a semicolon) must appear at the end of the list in order to terminate the **CASE** statement. We will be concerned with selector variables of type **INTEGER** at this time.

Listing 5.2

```
PROGRAM moonphase;
CONST
    dayphcorr  = 10;
    lencycle   = 28.3;
VAR
    daynumber,intphase              :INTEGER;
    startphase,phase,month,day,year :INTEGER;
    realphase,phasecorrection       :REAL;
BEGIN
    WRITE(OUTPUT,' *** Lunar Phase calculation program');
    WRITELN(OUTPUT,' ***');
    WRITELN(OUTPUT,' Enter the month/day/year:');
    READLN(INPUT,month,day,year);
    startphase := ((year-78) * 365) + dayphcorr ;
    CASE month OF
        1: daynumber:=1;
        2: daynumber:=32;
        3: daynumber:=60;
        4: daynumber:=91;
        5: daynumber:=121;
        6: daynumber:=152;
        7: daynumber:=182;
        8: daynumber:=213;
        9: daynumber:=243;
        10: daynumber:=274;
        11: daynumber:=304;
        12: daynumber:=334;
    END;    (*case*)
    startphase := startphase + daynumber + day;
    realphase := startphase / lencycle;
    intphase := TRUNC(realphase);
    realphase:=realphase-intphase;
    phase:=realphase * lencycle;
    CASE phase OF
    1,2,3,4,5,6,7        : WRITELN(OUTPUT,
                               'The moon is in its first quarter.');
    8,9,10,11,12,13,14   : WRITELN(OUTPUT,
                               'The moon is in its second quarter.');
    15,16,17,18,19,20,21 : WRITELN(OUTPUT,
                               'The moon is in its third quarter.');
    22,23,24,25,26,27,28 : WRITELN(OUTPUT,
                               'The moon is in its fourth quarter.');
    END;      (*case*)
END.        (*PROGRAM*)
```

The program of Listing 5.2 is designed to compute the phase of the moon. Two **CASE** statements are used, with different case selector lists.

The first **CASE** statement assigns a value to the variable **daynumber**, depending upon which integer you entered for **month**, two lines above. The format of the statement is:

CASE variable OF

or, in this example:

CASE month OF

If, for example, you had entered **10** as the month, the **CASE** statement would set the variable **daynumber** to **274**, since the first day of the tenth month happens to be day 274 of the year. (This program does not take leap years into account.) The day of the month that you earlier entered is then added to 274, to take into account any days that have passed since the first day of the month.

The second **CASE** statement selects a particular message for output, depending upon the resulting value calculated for the variable **phase.**

Now, we can take a broad look at the whole program. Calculations are based on a known starting phase of the moon at some past day and year. The first calculation for **startphase** yields the number of days since this known starting date as a function of the number of years, corrected for the starting phase of the moon.

The remainder of the calculations simply adjust the result value now stored by **startphase** to yield the whole number of days since the known starting phase, then divide the resultant number of days by the lunar cycle length in days.

Notice that expressions consisting of mixed real and integer arithmetic are used throughout the calculations. (Examples are: **realphase:=realphase-intphase** and **phase:=realphase * lencycle.**) Study of Table 4.2 will verify their validity.

Notice that the value of **realphase** is used as an argument for the **TRUNC** function. **TRUNC** is a predefined function available in Pascal that will truncate a real number (that is, drop of its decimal portion) and store the result in an integer.

CHAPTER FIVE QUIZ

1. The ______________ statement is used to make a single or compound statement execute a specific number of times.

2. For successive iterations (repeats) of a loop that uses the __________ statement, the loop-control variable is either incremented by one or decremented by one.

3. The _______________ statement is used to select a statement to execute from a list of statements.

4. "**Downto**" and "**to**" are elements of the _______________ statement.

5. A(n) _______________ statement is used to terminate the **CASE** statement.

CHAPTER FIVE ACTIVITY

Run (or simply examine) the program of Listing 5.2, and answer the following questions:

1. If you enter **12** for the month, what value will be used for **daynumber** in the second **startphase** calculation? _______________

2. If you enter 5 for the month, what value will be used for **daynumber** in the second **startphase** calculation? _______________

3. If the **phase** calculation results in the value **21**, which message will be printed?

____

CHAPTER SIX

DECISION TESTING

In Chapter Five, you learned how to use the **CASE** statement to make a test that determined the flow of the program. However, it may become necessary to perform more complex tests than the **CASE** statement was intended for. Pascal has a powerful set of logical and relational operators that make such testing easy.

LOGICAL AND RELATIONAL OPERATORS

Logical and relational operators are used in evaluating an expression to return a value of **true** or **false.** This **true** or **false** value can then be used to determine program flow.

LOGICAL OPERATORS

Pascal's logical operators are **AND**, **OR**, and **NOT.** These operators can be used to evaluate variables of type **BOOLEAN.** (As you remember from Chapter Two, a variable declared to be of type **BOOLEAN** can hold the value **true** or **false.**) Logical operators can be used in expressions that include numeric values and other kinds of variables.

AND evaluates two expressions. If both are true, **AND** returns the value **true.** If one or both are false, the value **false** is returned. For example, if two **BOOLEAN** variables, **Monday** and **October**, both held the value **true**, then the operation:

Monday AND October

would return the value **true.**

OR evaluates two expressions and returns the value **true** if either is true. If neither is true, the value **false** is returned.

NOT changes a **BOOLEAN** value to the opposite value. (For the moment, you can think of **NOT** as being to a logical value as the negative sign is to a number.) For example, the expression:

Saturday AND NOT(School)

would be evaluated as true if **Saturday** was true and **School** was false. Although **NOT** is really a Boolean operator, its action may be more clearly understood with a numeric example:

5 + 3 = 8 AND NOT(5 + 3 = 9)

would be evaluated as **true.**

Among logical operators, **NOT** has the highest precedence (that is, it is evaluated first), with **AND** second and **OR** last. Parentheses may be used to change the natural order of precedence. When logical operators are used with mathematical expressions, the mathematical expressions have precedence. Mathematical expressions are evaluated in the order described in Chapter Four.

Below are some examples of legal expressions using the logical operators. The letters "a", "b", and "c" below may be replaced by the variable names or by mathematical expressions.

Expression	Evaluated as
a AND b	**a AND b**
a OR b	**a OR b**
a AND NOT b	**a AND (NOT b)**
a AND b OR c	**(a AND b) OR c**
a AND (b OR c)	**a AND (b OR c)**
a OR b AND c	**a OR (b AND c)**
a OR NOT b	**a OR (NOT b)**

RELATIONAL OPERATORS

It is often necessary to compare several variables for equality in an expression to determine the flow of control. This may be accomplished by relational testing. Like the logical operators, the relational operators can also return a value of **true** or **false.** There are six relational operators in Pascal:

Operator	Meaning
=	equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
<>	not equal to

All six relational operators have equal precedence. They are evaluated after arithmetic operators, but before the logical operators. For example, the expression:

5 + cost > 10 OR 5 + cost < 500 AND number <> 0

is evaluated as follows:

((5 + cost) > 10) OR (((5 + cost) < 500) AND (number <> 0))

Precedence may be altered using parentheses.

If the relational test fails, the value of **false** is returned by the expression. If the test succeeds, then **true** is returned.

As shown in the example above, variables and numbers may be mixed in a relational expression. However, the arguments for a relational operator must be of the same type. For example, for the expression **number <> 0** to be valid, **number** must be declared as a **REAL** or **INTEGER** variable.

THE IF THEN ELSE, WHILE, AND REPEAT STATEMENTS

THE IF THEN ELSE STATEMENT

The logical and relational operators are frequently used with the **IF THEN ELSE** statement, which takes this basic form:

```
IF condition THEN
   statement
ELSE statement;
```

The **IF** is followed by a condition. If the condition is met, **THEN** the computer executes the statement following **THEN**. If the condition is not met, the computer skips to the **ELSE** and executes the statement following the **ELSE**. Notice that a semicolon must not precede the **ELSE** in the **IF THEN ELSE** statement.

Listing 6.1 provides a simple example of the **IF THEN ELSE** statement used with the logical operator **AND**. In this case, the variables **Monday** and **October** are assigned the value **true** before being evaluated by **AND**:

Listing 6.1

```
PROGRAM testIF;
VAR
Monday,October:BOOLEAN;
BEGIN
   Monday:=true;
   October:=true;
   IF October AND Monday THEN
      WRITELN(OUTPUT,'It is October and Monday')
   (*notice no semicolon after the previous statement*)
   ELSE WRITELN(OUTPUT,'Date unknown.');
END.
```

This program will print the message "**It is October and Monday**", since the conditions of the **IF** have been met.

Examine the program fragment of Listing 6.2, where **income** is a variable of type **INTEGER** and **president** is a **BOOLEAN** variable. Notice how variable of different types, a relational operator (>), and two logical operators (**AND** and **NOT**), are used together to form a single expression. Notice also that a compound statement is used after **THEN**:

Listing 6.2

```
   IF (income > 32000) AND NOT(president) THEN
      BEGIN
      WRITELN(OUTPUT,'You are being audited by the IRS.');
      WRITELN(OUTPUT,'Please justify your deductions.');
      END;
```

(If an **ELSE** followed the **END** in Listing 6.2, then the semicolon after **END** would have to be removed.)

The value of the expression in Listing 6.2 will be **true** if the integer value of **income** is greater than **32000** and the boolean value of **president** is **false**. (When the value of **president** is **false**, the **NOT** operator will reverse its value to **true**.)

THE WHILE STATEMENT

WHILE causes a statement (or compound statement) to execute as long as the condition named in the **WHILE** is **true**. The condition may be the value of a boolean variable or the boolean result of some expression. Some computation inside the loop should be able to change one of the variables in the condition, causing the test to fail at an appropriate time. Examine the program fragment of Listing 6.3, where **cnt**, **cost**, and **unitprice** are **INTEGER** variables and **underbudget** is a **BOOLEAN** variable:

Listing 6.3

```
cnt:=0;
underbudget:=true;
WHILE (cnt < 20) AND (underbudget) DO
   BEGIN
   cnt:=cnt + 1;
   cost:=cnt * unitprice;
   IF (cost > 200) THEN underbudget:=false;
   END;
```

This example will execute as a conditional loop, rather than executing a predetermined number of times with the **FOR** statement. When **cnt** becomes 21 or **cost** exceeds 200, the loop will terminate. (Note that the test **cost** > 200 could have been put in the **WHILE** expression just as easily.)

THE REPEAT STATEMENT

The **REPEAT** statement is similar to the **WHILE** statement. **REPEAT** repeats a statement or compound statement until an expression becomes true. The major difference between **WHILE** and **REPEAT** is that the **REPEAT** statement will always execute at least once because the test occurs at the end of the loop. Beginning programmers should use **REPEAT** with care (there may be times when you don't want the loop to execute at all). An example of **REPEAT** is as follows:

Listing 6.4

```
cnt:=0;
underbudget:=true;
REPEAT
   cnt:=cnt + 1;
   cost:=cnt * unitprice;
   inventory:=inventory + 1;
   IF (cost > 200) then underbudget:=false;
UNTIL(cnt>=20) OR NOT(underbudget);
```

Notice that the test was changed to use the **OR** operator instead of the **AND** operator. This is simply due to the different context of the two statements. No **BEGIN** or **END** is required. The statement(s) to be executed are simply placed between the **REPEAT** and **UNTIL**.

What happens to this loop if the initial value of **unitprice** is greater than 200? The loop will terminate on the first iteration, after it alters the value of **inventory**. This might not be the desired result and could cause an illegal entry into the inventory. In that case, the **WHILE** statement would have been a better choice.

CHAPTER SIX QUIZ

1. The logical operators in Pascal are: ____________, ____________ and ____________ .

2. The **AND** operator will return a value of ______________ if the value of the both expressions it is evaluating is true.

3. The **OR** operator will return a value of ____________ if one of the expressions it is evaluating is true.

4. The ____________ operator will reverse the value of a boolean variable or expression.

5. The **IF** statement will execute the **ELSE** portion of the statement if the value of the expression is _______________.

6. The **WHILE** statement will execute as long as the boolean result of the expression is _________________.

7. The _________________ statement will execute a statement or compound statement **UNTIL** the expression at the end of the statement is evaluated to be true.

CHAPTER SIX ACTIVITY

Tell whether the boolean value returned in each of the following situations is true or false:

1. **Tuesday:=true;**
   cornedbeef:=true;

   Tuesday AND cornedbeef

2. **Saturday:=true;**
   school:=false;

   Saturday AND NOT(school)

3. **5 > 10 OR 5 = 10**

CHAPTER SEVEN

PROCEDURES AND FUNCTIONS

One of the strengths of Pascal is that it promotes modularity. A modular program is organized into sections, each of which performs a specific function. (Programs in many other languages can be written as one large block of continuous statements.)

One of the reasons that Pascal programs may have a high degree of modularity is that the language was designed with procedures and functions in mind.

USING PASCAL PROCEDURES

Procedures may be thought of as complete sub-programs that have data passed to them as needed. They can be thought of as building blocks of the Pascal program. In Listing 7.1 on page B-40, notice that the procedure **readn** is structured much like a complete program.

The purpose of the program of Listing 7.1 is to read a positive integer from the file **INPUT** and to check for illegal entries. The procedure **readn** represents the typical use for a procedure, as it might be called (used) several times, from different points in the program. Since there is only one copy of this procedure in memory, no matter how many times it is called, considerable memory space is saved. (Further, a procedure's variables do not occupy memory space until the procedure is actually called.)

GENERAL CHARACTERISTICS OF A PROCEDURE:

The procedure name is followed by a parameter list. A parameter list is a list of values that are passed from one program unit to another. In this case, two of the variables declared in the main program (**number** and **legal**) are passed to the procedure. Then, the procedure goes on to declare some additional variables of its own in the procedure's **VAR** section.

The procedure differs from a compound statement by having a name and variables and constants of its own. It may contain one or more compound statements. However, like the compound statement, the procedure has a semicolon rather than a period after its **END** statement.

Unlike the compound statement, the procedure can be executed several times from different parts of the program.

Listing 7.1

```
PROGRAM INSTRUCTIONAL;

VAR
   number    :INTEGER;
   posnumber:INTEGER;
   legal     :BOOLEAN;

PROCEDURE readn ( VAR number: INTEGER; VAR legal: BOOLEAN );
(* The purpose of this routine is to read       *)
(* a positive number from a file in a           *)
(* character format and convert it to an integer*)
(* format.                                      *)
VAR
   loopcontrol,forcntr,inc:INTEGER;
   string                 :ARRAY [1..72]OF CHAR;
BEGIN
   FOR loopcontrol :=1 to 72 DO string[loopcontrol]:=' ';
   loopcontrol :=0;
   WHILE NOT EOLN(INPUT) DO
      BEGIN
      loopcontrol := loopcontrol + 1;
      READ(string[loopcontrol]);
      IF(string[loopcontrol]=' ')THEN
         (* Remove all leading blanks from array *)
         loopcontrol:=loopcontrol - 1;
      END;
   number:=0;
   inc:=1;
   FOR forcntr :=loopcontrol  DOWNTO 1 DO
      BEGIN
      number:=number+((ord(string[forcntr])-ord('0'))*inc);
      inc:=inc*10;
      END;
   IF (number < 0) THEN
      BEGIN
      legal := false;
      WRITELN('* Error - Illegal entry. Try again. ');
      END
   ELSE legal:= true;
   READLN(INPUT);
END;              (*procedure readn*)

BEGIN
legal:=false;
WHILE NOT legal DO
   BEGIN
      WRITELN('Enter any positive number:');
      READN(posnumber,legal);
      IF legal THEN writeln('The number is ',posnumber)
      ELSE WRITELN('ILLEGAL NUMBER')
   END;

END.
```

USING VARIABLES IN A PROCEDURE

Variables in a complex program are termed local or global, depending on how they are used.

Local variables are used only within a procedure (or function, yet to be introduced), or only within the main program. Local variables are declared in the **VAR** section of the unit where they are used. Variables from the main program which are redeclared within a procedure are considered to be separate, local, variables.

Global variables are used in the main program and also in one or more sub-programs.

There are several ways to use global variables. The first, non-recommended way, is simply to use the variable in the sub-program, without declaring it again. This is perfectly legal, since the variable has been declared once already in the main program. The trouble lies in the fact that: If the sub-program is called by the main program and the sub-program changes the value of any global variable, then the global variable will have the changed value when the main program again takes control. Global variables of this kind should be kept to a minimum, to avoid any accidental changes in their values.

BETTER WAYS TO USE GLOBAL VARIABLES

An alternate method of using global variables within a procedure is to pass them as parameters in a parameter list. This allows different variables to be passed at different times, and makes the use of the global variable much more visible in the program. The parameter list is placed immediately following the name of the procedure in the **PROCEDURE** declaration. A variable may be passed in the parameter list by one of two methods: by reference or by value.

When a variable is passed by reference, the actual variable with its value is passed to the procedure. If the procedure then changes the value of the variable, that value is changed in the entire program.
The format for passing a variable by reference is:

PROCEDURE procname (VAR varname1:TYPE; VAR varname2:TYPE);

When a variable is passed by value, what is passed is a copy of that variable's value. If the procedure alters the value, the value in the rest of the program is not changed. Parameters passed by value will prevent unwanted changes in a variable value by the called procedure. The format for passing by value is:

PROCEDURE procname (varname1:TYPE; varname2:TYPE);

Note that if the word **VAR** does not precede the variable's name, the variable is automatically passed by value.

It is perfectly legal for the same parameter list to contain some variables passed by reference and some variables passed by value. For example:

```
PROCEDURE example (date:INTEGER; VAR profit:REAL; cost:REAL);
```

By adhering to the convention of passing variables to a procedure, the programmer has an easier time determining how procedures alter external variables, and unwanted side-effects are minimized. Certainly global variables do have use in Pascal programs, but many new Pascal programmers have a tendency to over-use them.

CALLING A PROCEDURE

A procedure may be called from anywhere in the body of the program. In the call must be the procedure name, followed by a parameter list in parentheses. For example:

```
         procname (varname1, varname2);

or       example (date, profit, cost);
```

USING PASCAL FUNCTIONS

Another block in Pascal similar to the procedure is the <u>function</u>. Its internal structure is the same as the procedure, with **CONST** and **VAR** sections optional. The purpose of a function is similar to that of a procedure. Unlike the procedure, the function may not stand alone as a statement. Instead, the function is used like a variable, in an expression. Consider the following program:

Listing 7.2

```
PROGRAM functiontest;
VAR
   num:INTEGER;

FUNCTION ABS(number:INTEGER):INTEGER;
BEGIN
IF (number < 0) THEN ABS:= -number
   ELSE ABS:=number;
END;

BEGIN
   num:= -30;
   num:= ABS(num);
   WRITELN(' the absolute value of num =  ',num);
END.
```

Between the **BEGIN** and **END** statement under the function declaration is a definition of what **ABS** does. The result returned by the function named **ABS** is declared to be of type **INTEGER**.

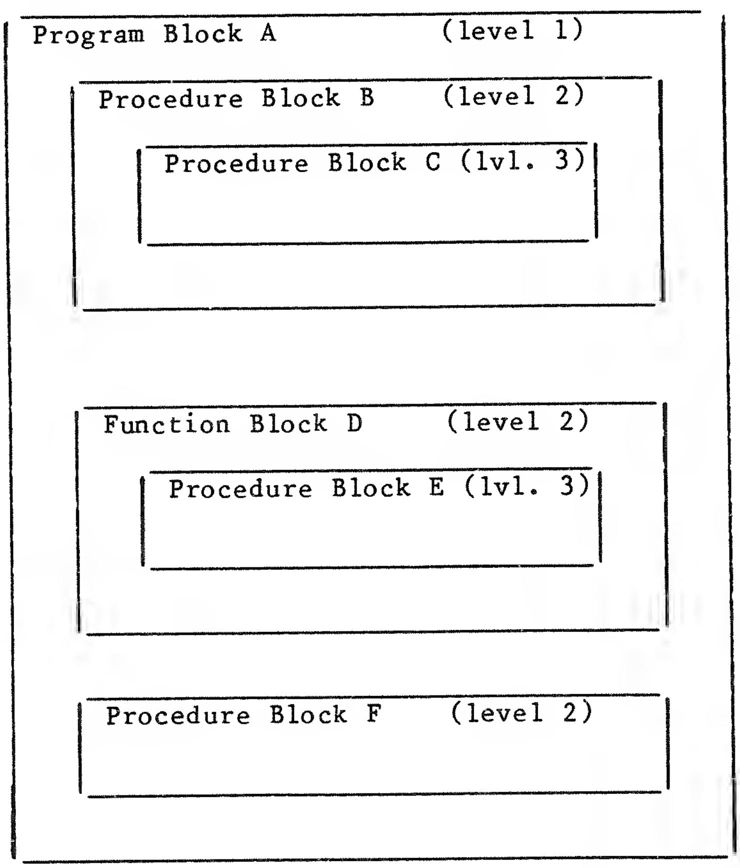
The result of any function must be used (like a variable) in an expression or assignment statement. In this case, the function's calculated value is transferred back to the calling program by an assignment statement that assigns the value to an identifier of the same name as the function name. The assignment statement **num:=ABS(num);** is needed. It would not be valid to simply say **ABS(num)**.

(The **ABS** function is already predefined in Pascal, although **ABS** is not a reserved word. This doesn't affect the example.)

ADVANCED PROGRAM STRUCTURE

Pascal is a <u>block-structured language</u>. This means that a program is constructed in a block-like manner. At a minimum, a program consists of one block. More blocks are created through the use of procedures and/or functions by placing them inside this outermost program block.

The term for this process is "<u>nesting</u>". The rule for nesting is that a block may lie entirely within another block, but blocks do not overlap in any other way. A level of nesting can be assigned to each block of a program. This block structure can be represented pictorially by the following diagram.



This block structure provides illustration for our earlier discussion of local and global variables. A global variable is declared in an outer enclosing block and is used in more than one block. A local variable is declared in an inner block and is used only in that block.

SCOPE RULES

Scope rules are the rules that govern accessing of variables, types, and constants, in a nested program. In general, these identifiers may be accessed in the program or procedure in which they are declared, or in any procedure declared within that procedure. All identifiers, including types, constants, variables, and procedure declarations, have scope (boundaries on their use).

If an identifier is redeclared within its scope (area), the outer definition becomes inaccessible within the scope of the inner definition. (Remember that you were earlier told that redeclaring a variable from the main program in a procedure would cause the program to view the variable as a separate entity within that procedure.) In the example below, declaration of **b** as an **INTEGER** within the inner procedure causes all references to **b** in that procedure to refer to the local variable -- not the **BOOLEAN** variable **b** in the main procedure. Because **i** is not redeclared, it continues as the global variable **i**.

Listing 7.3

```
PROGRAM globals;
VAR
i:INTEGER;
b:BOOLEAN;

PROCEDURE inner;
VAR
   b:INTEGER;
BEGIN
   b:=i + 25;
   i:=i + i;
END;

BEGIN
   i:=0;
   writeln(i);
END.
```

Pascal requires that all identifiers be declared before they are used. If the declaration of an identifier has not been encountered in the text of a program, then the identifier is considered undefined. This rule allows some procedures to call other procedures that cannot in turn call them.

A procedure can be called from the body of the block declaring it, from the procedures declared within it, and from procedures declared within the same block. However, if procedure A is declared before procedure B in the same block, then procedure B can call A, but procedure A cannot call B. This is because the declaration of B had not yet been encountered in the source text (program listing) when the body of procedure A was being compiled.

The above "visibility" restriction (A cannot "see" B) can be avoided with the use of **FORWARD** declarations. In a **FORWARD** declaration, the procedure is declared near the beginning of an outer program, but the body of the procedure is replaced by the word **FORWARD.** The actual body of the procedure is supplied later. If all procedures within a block are declared in this way, then any one of them can call any other.

Listing 7.4 provides an example of how **FORWARD** works. Note that the parameter list must be provided with the first declaration of each function or procedure. Later declarations of the function repeat the parameter list as a comment for documentation purposes. If the parameter list is repeated, it must be as a comment.

Listing 7.4

```
PROGRAM outer;
VAR
   i:INTEGER;
FUNCTION Distance(x1,x2:INTEGER):INTEGER; FORWARD;
FUNCTION ABS(tvalue:INTEGER):INTEGER; FORWARD;

FUNCTION Distance (*(x1,x2:INTEGER):INTEGER*);
BEGIN
   distance:=ABS(x2 - x1);
END;

FUNCTION ABS(*(tvalue:INTEGER):INTEGER*);
BEGIN
   IF tvalue < 0 THEN ABS:= -tvalue
   ELSE ABS:=tvalue;
END;

BEGIN
   WRITE('DISTANCE = ',Distance(8,2));
END.
```

CHAPTER SEVEN QUIZ

1. ________________ and ________________ promote modularity in programs.

2. Data may be passed to procedures and functions through a ________________ list.

3. Blocks may be ______________ within other blocks.

4. Nesting affects the _______________ of blocks.

5. Parameters may be passed by value or _______________ .

6. When a variable is passed by _________________ , what is actually passed is a copy of the variable.

7. When a variable is passed by reference, the keyword _____________ must precede it in the parameter list.

CHAPTER SEVEN ACTIVITY

Look back at Listing 7.1 and complete the following statements about the program.

1. The variables ________________ and ________________ are passed to the procedure **READN.**

2. The variables ________________, ________________, ________________, and ________________ are local to the procedure **READN.**

3. As the program is currently written, which variables may not be accessed by the main program?

   ____

4. Variables are passed to procedure **READN** by value or by reference?

   ____

CHAPTER EIGHT

ADVANCED DATA TYPES

ARRAYS

The array is a variation on the data types already discussed. Sometimes many variables of a particular type are needed. For example, if you needed seventy-two variables of the type **CHAR** to represent a series of characters to be input, you could declare all seventy-two separately. However, this would be time-consuming. Furthermore, accessing the individual variables would be confusing, since each would have a different name.

The simple answer to this problem is the data type **ARRAY**. You may declare an array as :

```
arrayname:ARRAY (.1..n.) of TYPE;
```

where **TYPE** is any of the data types covered in Chapter Two. (User-defined data types, covered later in this chapter, can also be used.) The **n** is the number of variables desired. For example, the following array would store a string of twenty-six characters:

```
alphabet:ARRAY (.1..26.) of CHAR;
```

NOTE: Standard Pascal uses brackets **[1..n]** instead of the parentheses and periods. On Models III and 4, "(." replaces "[" and ".)" replaces "]" -- or you can create "[" by pressing **CLEAR 1** and "]" by pressing **CLEAR 2**.

INPUT/OUTPUT USING ARRAYS

Let's take a look at the example program of Listing 8.1. to see how the example might be used:

Listing 8.1

```
PROGRAM onedimarray;
VAR
   alphabet:ARRAY (.1..26.) of CHAR;
BEGIN
   WRITELN('Enter the alphabet.');
   READLN(INPUT,alphabet);
   WRITELN(OUTPUT,alphabet);
   WRITELN('Program complete.');
END.
```

If the input/output of this program is directed to the video display, the program will display the prompt **"Enter the alphabet."** At this point, the user types in a twenty-six character string. Any characters typed will be considered as part of the string.

Only at the end of the string does the user press **ENTER.** The characters are then input to the array, left justified. If the length of the input character string is less than the length of the array, the remaining storage positions in the array will contain blanks. If the input string is longer than the array, any extra characters will be ignored.

The program of Listing 8.1 will then output the string back to the video display. In TRS-80 Pascal, a <u>one-dimensional</u> array like this one -- an array of characters in one row -- may be input and output by a single **READ** and **WRITE.**

ACCESSING INDIVIDUAL COMPONENTS OF AN ARRAY

Elements of an array may be assigned to variables of the same type as the array. For the example we have been considering, this would be a variable of type **CHAR**:

```
VAR
   alphabet:ARRAY (.1..26.) of CHAR;
   letter:CHAR;
```

To assign to **letter** a particular component of this array, you would use the component variable name **letter**, plus the array name and a subscript that denotes the position, in the array, of the component you want. The assignment statement

```
      letter:=alphabet(.4.);
```

would assign to **letter** the value of the fourth element in the array line. (If the user entered the alphabet correctly, **D** would be the fourth element.)

Components of an array may be accessed without use of a component variable. For example: **WRITE(alphabet(.4.));.**

PACKED ARRAYS

Any array may be declared with the word **PACKED** as a prefix. **PACKED** tells the compiler to store the data elements as efficiently as possible.

In some versions of Pascal, you may not pass elements of packed structures by reference to procedures or functions, and packed elements may not be used as arguments in **READ** statements. In TRS-80 Pascal, there are no such restrictions.

ARRAYS WITH MORE THAN ONE DIMENSION

So far, we have considered one-dimensional arrays. However, arrays in Pascal may have several dimensions. Suppose that you have several strings of characters for input, and you want to store every character string. This can be done easily by declaring an array as follows:

arrayname : ARRAY (.1..n,1..z.) of TYPE;

The **READ** command will not input an entire multidimensional array automatically. However, **READ** will input the one-dimensional sub-arrays, one at a time. Listing 8.2 demonstrates use of a two-dimensional array:

Listing 8.2

```
PROGRAM arrayIO;
VAR
   I          : INTEGER;
   stringl    : ARRAY [1..5,1..72] OF CHAR;
BEGIN
   FOR I:= 1 TO 5 DO
      BEGIN
      WRITELN(OUTPUT,'Enter command line ',I);
      READLN(INPUT,stringl[I] );
      END;
END.
```

This program will prompt the user for five different command lines. Each time the loop is executed, an individual sub-array is loaded into the array.

Since the elements of this array are of type **CHAR**, any operations that can be performed on a simple variable of type **CHAR** may be performed on an element of this array.

Remember that an array may be of any type, such as **BOOLEAN**, **INTEGER**, or any user-defined data type, including **ARRAY**. For most simple data types, the name of the type may be substituted for the bounds, which are declared as constants in the array declaration. In this case, the number of elements in the data type is the number of elements in the array.

USER-DEFINED DATA TYPES

The data types explained so far are pre-defined by Pascal. But Pascal also allows you to define new data types at will. These defined types have names chosen by the programmer and are declared in the program's **TYPE** section. Once declared, they may be used wherever declared data types are allowed.

Let's consider an example of when user-defined data types might be desirable. Suppose that the programmer is manipulating an integer variable in BASIC that may take on one of four values: 1, 2, 3, or 4. The numbers may in fact represent the colors red, green, blue and orange. When the value is 1, a message is written to the terminal saying that the color red is being processed, and so on. This process is typically known as decoding information from a variable's value.

Needless to say, when BASIC programs get very long, it is difficult to determine their flow because of this decoding and encoding of information. A simpler way would be to declare a variable that could directly take on the value of **red**, **green**, **blue** or **orange**. Then tests could be performed to see if the value of the variable is **red**, etc. Program logic would be much clearer and easier to follow. In fact, this is exactly what the following program does.

Listing 8.3

```
PROGRAM usertypes;
TYPE  difcolor = (red, green, blue, orange);
VAR
   color : difcolor;
BEGIN
   color := red;
   REPEAT
      CASE color of
         red    : WRITELN(OUTPUT,' The color is red');
         green  : WRITELN(OUTPUT,' The color is green');
         blue   : WRITELN(OUTPUT,' The color is blue');
         orange : WRITELN(OUTPUT,' The color is orange');
         END;
   color := SUCC(color);
   UNTIL( color = orange );
END.
```

ENUMERATED USER-DEFINED TYPES

Program 8.3 illustrates an enumerated user defined type, **difcolor.** A type is said to be enumerated if a list of possible variable values is given in the type declaration.

The predefined function, **SUCC** is frequently used with user-defined variable types, to increment to the next possible value. In a simple program using an integer variable, this could be accomplished by adding 1 to the variable, but this would not make sense with a user-defined type.

User-defined enumerated data types may not have their values written out. Program 8.3 gets around this limitation by selecting a literal message from a **CASE** list, depending on the variable's value.

SUBRANGE TYPES

A variable may assume a value that is a subset of some predefined type. In this case, it may be declared to be a subrange type.

For example, the type **INTEGER** may represent any whole numbers between -32,768 and 32,767. A subrange of the type **INTEGER** might be declared as follows:

```
byte = 0..255;
```

In this example, any variable of the type **BYTE** may take on a value from 0 to 255. Any operations that may be performed on the original predefined type may be performed on the subrange type.

A subrange type may also be the subrange of any user-defined simple type. Listing 8.4 demonstrates a series of valid subrange declarations:

Listing 8.4

```
PROGRAM subrange;
TYPE
   baddate          = 1900..1903;
   uppercaseletters = 'A'..'Z';
   lowercaseletters = 'a'..'z';
   digits           = '0'..'9';
   xaxis            = -100..100;
VAR
   testyear          : baddate;
   upperletter       : uppercaseletters;
   lowerletter       : lowercaseletters;
   digit             : digits;
   xpoint            : xaxis;
BEGIN
END.
```

Named subrange types are useful when the programmer wants to increase program readability by clearly identifying the data differences between specific variables. Also, storage space may be saved by using subrange variables. The storage required for a subrange variable depends on the range of values. This consideration may be important when you are building large data structures to be implemented on microcomputers. For example: the type **BYTE** defined above only requires one byte (8 bits) of storage while an **INTEGER** requires two bytes (16 bits).

RECORD DATA TYPES

So far, the only structured data type examined has been the array. The array is an excellent mechanism for storing large amounts of data of the same type. In the example, a series of text strings was efficiently stored using arrays of **CHAR**, and any individual character was easily accessible.

However, you may want to group variables of different data types. For example, suppose that a business wanted to keep the following information about each of its customers:

Name
Customer category
Mailing address
Telephone number
Dollars spent in store
On catalog circulation list

In languages like BASIC, the usual way to maintain this information would be multiple arrays containing encoded information. This is not the case in Pascal. You can build a record which can store all of the above information in a clear and concise format. Furthermore, you may declare arrays to be of the user-defined type **RECORD**.

A record in Pascal is a predefined data structure which is composed of component variables. These components may be variables of any Pascal predefined, or user-defined data types. The purpose of a record is to group variable information into logical entities, such that any particular component may be operated on. The entire record may also be referenced as a whole.

Program 8.5 on the following page is an example of how the business record would be declared in Pascal.

Listing 8.5

```
PROGRAM database;
TYPE
   custmrcategory = (business,individual);
custmrecord = RECORD
   custmrtype    : custmrcategory;
   address       : PACKED ARRAY[1..72] OF CHAR;
   telephone     : PACKED ARRAY[1..15] OF CHAR;
   expenditures  : REAL;
   cataloglist   : BOOLEAN;
   END;
VAR
   custmr       : custmrecord;
   custmrlist   : ARRAY[1..100] OF custmrecord;
   index        : INTEGER;
   ans          : CHAR;
PROCEDURE custmrinp( VAR custmr:custmrecord);
VAR custyp : CHAR;
BEGIN
   WRITELN('* Enter customer type: (business/individual)');
   READLN(custyp);
   IF(custyp='I')THEN
      custmr.custmrtype:=individual
   ELSE custmr.custmrtype:=business;
   WRITELN('* Enter address:');
   READLN(custmr.address);
   WRITELN('* Enter telephone number:');
   READLN(custmr.telephone);
   WRITELN('* Enter expenditure in dollars:');
   READLN(custmr.expenditures);
   WRITELN('* Want on catalog circulation list:  (true/false)');
   READLN(custmr.cataloglist);
END;
BEGIN
   index:=0;
   ans:='N';
   WRITELN('** BUSINESS XYZ CUSTOMER RECORD PROGRAM **');
   WHILE (ans <> 'S') DO
      BEGIN
      index:=index+1;
      custmrinp(custmrlist[index]);
      WRITELN('* MORE CUSTOMERS (STOP/CONTINUE)');
      READLN(ans);
      END;
END.
```

The outer shell that must enclose record type declarations is of the form:

```
typename = RECORD
END;
```

For example:

```
custmrecord = RECORD
END;
```

The component field (variable) declarations reside between **RECORD** and **END;**. The field declarations are defined in the same way as they would be in the **VAR** section of the program.

In Listing 8.5, the user-defined record name is **custmrecord.** The component field declarations: **custmrtype**, **address**, **telephone**, **expenditures**, and **cataloglist** are defined exactly the same way as the program variables are in the **VAR** section. All of the field components belong to the data type **custmrecord.**

Since **custmrecord** may be treated like any other user-defined type, we can next declare a variable to be of type **custmrecord** in the **VAR** section of the main program.

The difference between a record and the simple user-defined data types is that there are component fields in a record that are really variables themselves. In example 8.5, the variable **custmr** is of a record type. When referring to **custmr** in expressions, that variable name references the entire record. To access the component field **expenditures**, you would prefix **expenditures** with the record variable name, **custmr** and a period. For example:

```
custmr.expenditures:= 99.95;
```

Values in one record may be transferred to another record. For example, if a variable of type **RECORD** named **excustomer** had been declared, the following statement would set all component fields in **excustmr** to the component fields in **custmr**:

```
excustomer:=custmr;
```

Variables of type **RECORD**, and their associated component fields, obey the same rules for use as all other types of variables.

The program of Listing 8.5 performs record I/O using the predeclared text files **INPUT** and **OUTPUT**. Notice that the **READ** and **WRITE** statements use record component fields as arguments. **READ** and **WRITE** behave as though the component fields were variables declared in the **VAR** section.

THE WITH STATEMENT

Use of records may cause segments of the program that reference them to become long and tedious. (Remember that every time a component field is referenced, the record name must precede it.) The **WITH** statement can be used to simplify the process of accessing component fields.

Examine the following procedure, which could have been used in Listing 8.5:

Listing 8.6

```
PROCEDURE custmroutput(VAR custmr:custmrecord);

BEGIN
   WRITELN('** CUSTOMER OUTPUT RECORD FOR BUSINESS XYY **');
   WITH custmr DO
      BEGIN
      IF(custmrtype=business)then
         WRITELN  ('Customer type     :  Business')
      ELSE WRITELN('Customer type     :  Individual');
      WRITELN     ('Address           :  ',address);
      WRITELN     ('Telephone         :  ',telephone);
      WRITELN     ('Expenditures      :  ',expenditures);
      WRITE       ('Circulation list : ');
      IF (cataloglist)THEN WRITELN('Yes')
      ELSE WRITELN('No');
      END;
END;
```

The **WITH** statement eliminates the need to use the record name as a prefix when referencing components of the record. The scope of the **WITH** is one statement, which in this case is a compound statement.

FILE OF TYPE

INPUT and **OUTPUT** are examples of **TEXT** files in Pascal. **TEXT** files have been used for all of the examples so far. A **TEXT** file is predeclared by Pascal to be a special file of **CHAR**, with rules for performing I/O using **INTEGER**, **REAL** and **BOOLEAN** variables.

In TRS-80 Pascal, there are extensions to allow for performing I/O using **ARRAY** variables in text files. A **FILE OF <any known type>** may be declared in Pascal. Files of types other than text are primarily used for storing data which will be retrieved at some other time. For example, a **FILE OF custmrecord** could be defined in the type section. (**custmrecord** was defined in listing 8.5) A variable of type **custmrecord** could be written to this file. The important thing to remember is that an entire record may be written (or read), by one I/O statement. Component fields of this record may not be read or written individually to a file of records.

When I/O is performed with a **FILE OF <any type except text>**, no ASCII encoding or decoding of information takes place. Instead, the binary representation is used. This is not particularly useful when the I/O is directed to a terminal, but is effective for storing large amounts of information on disk media. The predeclared procedures **WRITELN** and **READLN** are not valid when performing I/O with a file of any type except **TEXT**, although **READ** and **WRITE** perform normally.

CHAPTER EIGHT QUIZ

1. If a large number of variables of the same TYPE need to be declared, the ______________ may be the correct data structure to use.

2. Arrays in Pascal may have one or more _________________.

3. New user defined _____________ may be declared in Pascal programs.

4. A(n) ________________ TYPE is defined by a list of identifiers given to be the different values allowable for a variable.

5. A ______________ TYPE is any user defined TYPE that is a sub-interval of another simple type.

6. A _______________ TYPE is used to logically group together data of different types.

CHAPTER EIGHT ACTIVITY

Using the program of listing 8.5 for guidance, write a short program that includes a record. Compile and run the program.

CHAPTER NINE

DYNAMIC DATA TYPES

All of the variable types discussed so far have been "static" in nature, meaning that their size had to be defined before the program was compiled or executed. For example, in program 8.5, the size of the array **custmrlist** has an upper bound of 100 entries. If more than 100 storage locations were needed to store the customer records, the array declaration would have to be changed in the source program, and the source code recompiled.

Most Pascal implementations on microcomputers limit the number of storage locations that may be declared in a program. This limitation is based on the size of the program and the type and number of variable declarations. Because of these memory restrictions, Pascal programmers should usually declare arrays and other data structures to be only as large as required.

Static variable declarations create problems in some programming applications. For example, suppose that in program 8.5 you wanted to keep a list of sales transactions for each customer along with each customer record. You could accomplish this by adding a component field (an array of transaction records) to each customer record. Then at any time you could access the sales transactions of customers. But, the number of sales transactions per customer would have to be limited to a preset number by the array declaration. The number of transactions per customer will probably vary, so use of a static variable in this case is not very efficient.

Pascal does allow for dynamic variable allocation at program execution time through the use of pointer variables and the Pascal predefined procedure **NEW**, as described below.

POINTER TYPE DECLARATIONS

Listing 9.1 illustrates how pointer variables are declared:

Listing 9.1

```
TYPE
    trxptr  = ^trxrec;

trxrec = RECORD
    nexttrx : trxptr;
    invoicenumber  : INTEGER;
    date           : ARRAY [1..10] OF CHAR;
    transprice     : REAL;
    partnumberlist : ARRAY [1..10] OF CHAR;
    END;

VAR
    trx : trxptr;
```

In the **TYPE** section of Listing 9.1, **trxptr = ^trxrec** declares the user-defined type **trxptr** to be a pointer to **trxrec**. That is, variables of type **trxptr** will point to a location in memory of the size required to store the record **trxrec**. (The character "^", created by pressing **CLEAR**, then 3, denotes a pointer in Pascal. The character "@" can be used in the place of "^".)

Notice that at the **^trxrec** point in the type declaration, **trxrec** has not yet been defined. In Pascal, declaring a pointer as a yet undefined type is valid.

In the **VAR** section, the variable **trx** is declared to be of the type **trxptr**. Therefore, the variable **trx** will point to a storage location for **trxrec**. This storage location may be requested anytime during program execution. The component field **nexttrx**, embedded within the record itself, is also a variable of type **trxrec**.

Pointer types to large data structures may be declared in a program with minimum memory space penalty until the procedure **NEW** is called during program execution. **NEW** performs the actual task of allocating storage for the data structures that are pointed to.

PROCEDURE NEW

The procedure **NEW** has one argument, which is a pointer variable. **NEW** allocates the amount of storage required by the data type that is associated with the pointer (in the example above, the data type is a record). Then **NEW** assigns the address of the allocated storage to the pointer variable. The pointer is then used to reference the allocated storage.

NEW does not increase the total amount of memory available, but dynamic allocation of memory does allow for better utilization of space. The variable used as an argument to the **NEW** procedure call must have been declared in the **VAR** section, and must be declared as a type that is a pointer to the actual data type.

The program segment of Listing 9.2 on page B-59 illustrates a few simple methods of using pointer variables.

Notice in Listing 9.2 that variables for which storage is dynamically created are not referenced like normal variables. The reason for this is that these variables actually have no identifiers of their own. A pointer must be used each time one of these variables is referenced. For example, **trx^.transprice** refers to the value of a component field in the record stored at the location pointed to by **trx**. When the contents of the storage location are being referred to, "^" follows the variable name.

When the pointer itself is being referenced, just the variable name is used. For example, note how, in Listing 9.2, the pointer variable **nexttrx** is set to the value of **trx** by the assignment statement **nexttrx:=trx;**.

Listing 9.2

```
PROGRAM dynamic;

TYPE
    trxptr = ^trxrec;

trxrec = RECORD
    nexttrx              : trxptr;
    invoicenumber        : INTEGER;
    date                 : ARRAY [1..10] OF CHAR;
    transprice           : REAL;
    partnumberlist       : ARRAY [1..10] OF CHAR;
    END;

VAR
    trx     : trxptr;
    nexttrx : trxptr;

BEGIN
    NEW(trx);
    trx^.invoicenumber:=2345;
    trx^.transprice   :=99.95;
    nexttrx:=trx;
    WRITELN('* Transaction invoicenumber : ',
           trx^.invoicenumber);
    WRITELN('* Transaction price         : ',
           trx^.transprice);
    DISPOSE(trx);
END.
```

PROCEDURE DISPOSE

Notice the call to the predefined procedure **DISPOSE** near the end of Listing 9.2. **DISPOSE** releases the storage area acquired in the **NEW** call. After the **DISPOSE**, the data stored at the dynamic memory location is effectively lost.

This is an important feature of Pascal. Careful use of **NEW** and **DISPOSE** can result in programs that dynamically grow and contract in memory size as needed, and efficiently manage the computer resources.

LINKED LIST

A linked list is the result of a programming technique that chains together a series of variables. A thorough discussion of linked list processing would entail several chapters, and is really a topic for a data structures book. It will be covered briefly here because it relates to dynamic memory management.

In example 9.1, the data type **trxrec** has a component field (**nexttrx**) which is a pointer to a storage area of the same type as itself. A pointer to another record node (storage area) may be stored in this field. In the record pointed to, a pointer to another record node could be stored, and so on. In this way, a series of record nodes may be linked together. The diagram of Listing 9.3 will help you visualize this list. The program of Listing 9.4 illustrates how the structure in Listing 9.3 could actually be built.

Listing 9.3

```
VAR
    headnode : trxptr;
```

headnode

```
                                  record number 1

trxrec = RECORD
    nexttrx: trxptr;
    invoicenumber  : INTEGER;
    date           : ARRAY [1..10] OF CHAR;
    transprice     : REAL;
    partnumberlist : ARRAY [1..10] OF INTEGER;
    END;
```

```
                                  record number 2

trxrec = RECORD
    nexttrx: trxptr;
    invoicenumber  : INTEGER;
    date           : ARRAY [1..10] OF CHAR;
    transprice     : REAL;
    partnumberlist : ARRAY [1..10] OF INTEGER;
    END;
```

```
                                  record number 3

trxrec = RECORD
    nexttrx: trxptr;
    invoicenumber  : INTEGER;
    date           : ARRAY [1..10] OF CHAR;
    transprice     : REAL;
    partnumberlist : ARRAY [1..10] OF INTEGER;
    END;
```

NIL.

The variable **headnode** is a pointer variable declared in the **VAR** section of the program. At some point in the program, a **NEW** procedure call could be made with **headnode** as its argument. **Headnode** would now be a pointer to the start of the list. Notice the word **NIL** at the end of the list. **NIL** is a reserved word in Pascal. This simply sets the pointer to an initialized value that may be tested for in looping statements. A word of caution when using pointers in Pascal: if a pointer variable has been declared, but not set to any value, there is no guarantee of its value. It will not necessarily be set to **NIL**.

Most Pascal implementations do not perform a runtime check for uninitialized values. Use of uninitialized pointers can lead to the program writing over itself in memory with execution becoming unpredictable. These kinds of programming errors will not show up at compile time, and can be extremely hard to find during program execution.

Listing 9.4

```
PROGRAM linkedlist(input,output);
TYPE
    trxptr = ^trxrec;
    textline       =  PACKED ARRAY [1..10] OF CHAR;
    trxrec = RECORD
        nexttrx: trxptr;
        invoicenumber  : INTEGER;
        date           : textline;
        transprice     : REAL;
        partnumberlist : textline;
        END;
VAR
    headnode, transnode    : trxptr;
                         I : INTEGER;
PROCEDURE readtrx(VAR trx:trxrec);
    (* The purpose of this routine is to prompt the user for *)
    (* the purchaser's trx record                            *)
BEGIN
    WITH trx DO
        BEGIN
        WRITELN(' ENTER INVOICE NUMBER:');
        READLN(invoicenumber);
        WRITELN(' ENTER DATE:');
        READLN(date);
        WRITELN(' ENTER TOTAL PURCHASE PRICE:');
        READLN(transprice);
        WRITELN(' ENTER PARTNUMBER(S) SEPARATED BY COMMAS:');
        READLN(partnumberlist);
        END;
END;    (* readtrx *)
```

Listing 9.4 (continued)

```
PROCEDURE writetrx(VAR trx:trxrec);
(* The purpose of this routine is to write the purchaser *)
(* trx entry                                             *)
VAR   I : INTEGER;
BEGIN
    WITH trx DO
        BEGIN
        FOR I := 1 TO 35 DO WRITE('*');
        WRITELN;
        WRITELN('INVOICE NUMBER         : ',invoicenumber);
        WRITELN('DATE                    : ',date);
        WRITELN('TOTAL PURCHASE PRICE    : ',transprice:10);
        WRITELN('PART NUMBER LIST        : ',partnumberlist);
        FOR I := 1 TO 35 DO WRITE('*');
        WRITELN;
        WRITELN;
        END;
END;    (* PROCEDURE writetrx *)

PROCEDURE listrxs( temptr : trxptr );
(* the purpose of this procedure is to traverse the linked*)
(* list attached to the argument pointer, writing the     *)
(* values of the trx records *)

VAR
    loctrx : trxrec;
BEGIN
    (* traverse trx linked list,writing trxs *)
WHILE (temptr <> NIL) DO
    BEGIN
        (* load the contents of localtrx with the *)
        (* contents of temptr  *)
    loctrx:=temptr^;
    writetrx(loctrx);
        (* set temptr to the next node in the linked list  *)
    temptr := temptr^.nexttrx;
    END;
END;   (*listransactions*)
```

Listing 9.4 (continued)

```
BEGIN     (* begin main program linkedlist *)
    (* initialize pointer that will always reflect the *)
    (* beginning of the list. *)
    (* this will set the end of the list to NIL during the first   *)
    (* pass through the FOR loop *)
headnode := NIL;
    (* read 3 trxs and link each new one to the beginning *)
    (* of the list *)
FOR I := 1 to 3 DO
    BEGIN
    NEW(transnode);
        (* insert the newnode in front of the old headnode *)
        (* link to the old headnode *)
    transnode^.nexttrx := headnode;
        (* make the newnode the new headnode *)
    headnode := transnode;
        (* load the actual data into the fields of the new node *)
    readtrx(transnode^);
    END;
    (* list all trxs entered *)
listtrxs(headnode);
END.  (*main program*)
```

CHAPTER NINE QUIZ

1. A "linked list" is a ____

   ____

   ____ .

2. The symbol ________ or ________ indicates use of the pointer data type.

3. The procedure ______________ releases the storage space that was allocated by the ______________ procedure.

4. True or false?: When you refer to a component field of a data structure for which storage was dynamically allocated, "+" follows the variable name.

   ____

5. True or false When the pointer itself is being referred to, only the variable name is used.

6. "**trx^.date**" refers to a ____________________________ of a data structure whose location is pointed to by the variable ____________.

CHAPTER NINE ACTIVITY

Using the program that you constructed for the Chapter Eight activity as a starting point, you may want to try developing a program that includes a record data structure for which storage is dynamically allocated. Portions of Listing 9.4 can provide helpful guidance.

For further information about the pointer data type, consult Chapter Five of the LANGUAGE REFERENCE GUIDE (pages D-37 through D-39).

Sets in Pascal have the same meaning as they do in the normal mathematical sense. If a group of objects is declared in set A, and a group of objects is declared in set B, a number of operations may be performed on these sets such as :

(i) membership and relational testing

(2) set arithmetic (union, intersection, difference)

In the case of Pascal, the objects are simply data values. These data values may be Pascal predefined or user defined. An example would be a **SET OF CHAR**, or a **SET OF digits** where digits is a user-defined subrange type of **CHAR.** Testing could be performed to see if the **SET OF digits** is in the **SET OF CHAR**, if desired.

The method of declaring set variables is :

VAR A,B : SET OF <type> ;

This means that **A** or **B** may contain from one to all of the data values declared by the type, however its membership is undefined until it is initialized like any other variable. In the body of the program, a set may be initialized to empty by:

A := [];

MEMBERSHIP TESTING

Once the set variables are initialized, a series of **BOOLEAN** relational tests may be performed. The relational operators are as follows:

set1 = set2	Set equality. If all members of first set are in the second set and all members of second set are in the first set, TRUE is returned.
set1 <= set2	Subset. If all members of first set are in the second set, TRUE is returned.
set1 >= set2	Superset. If all members of the second set are in the first set, TRUE is returned.
set1 <> set2	Set inequality. If all members of the first set are in the second set, and all members of the second set are in the first set, FALSE is returned.

Individual element membership may be tested by using the **IN** operator. If a variable had been declared of the same type as the base set type, the **IN** operator may be used to check for set membership. An example would be:

Listing 10.1

```
TYPE
    DIGITS = '0'..'9';
VAR
         DIGIT  : SET OF DIGITS;
         D      : CHAR;

    BEGIN
       D := 'a';
       DIGIT := ['0'..'9'];
       IF(D IN DIGIT)THEN DO (*action*);
       IF(D='0')OR(D='1')OR(D='2')OR(D='3')OR(D='4')OR(D='5')
         OR(D='6')OR(D='7')OR(D='8')OR(D='9') THEN DO (*action*)
```

The two **IF** statements in the above program segment are equivalent. Notice that the equivalent **IF** statement using sets is a more concise and readable statement. This represents a simple use for sets for the average programmer.

SET ARITHMETIC

There are three set operators in Pascal. Each requires two arguments. Arguments should be sets of the same base type, and the result will be of the same type. The operators are: **+**, *****, and **-**. When arithmetic operations are performed on two sets, these sets should be of compatible types (that is, they should be of the same base type, or both subranges of the same base type, or one must be a subrange of the other).

A + B	Gives the union of A and B
A * B	Gives the intersection of A and B
A - B	Gives the difference of A and B.

The segment program of Listing 12.2 illustrates set operator use.

Listing 12.2

```
PROGRAM TESTSET;
VAR
    DIGITS,LETTERS,LOWERCASE,UPPERCASE : SET OF CHAR;
    ALPHANUMERIC, ALPHA                : SET OF CHAR;
    D  : CHAR;
BEGIN
    D := '1';
    DIGITS := ['0',,'9'];
    LOWERCASE := ['a'..'z'];
    UPPERCASE := ['A'..'Z'];
    LETTERS := LOWERCASE + UPPERCASE;
    ALPHANUMERIC := LETTERS + DIGITS;
    ALPHA := ALPHANUMERIC - DIGITS;
    IF ( D IN ALPHANUMERIC * DIGITS ) THEN
       WRITELN( 'PUNT' );
END.
```

CHAPTER TEN QUIZ

1. True or false?: A set may contain any or all of the values declared by its type, or may be empty (a null set).

   ____

2. True or false?: Applying the set operator "*" to two sets results in a third set that contains all of the members that are in one or both of the sets.

   ____

3. The two arguments of a set operator should be of compatible ________.

4. Two sets are of compatible ____________ if:

   a. Both sets were declared as being of the same ____________.

   b. One set is a ______________ of the other set.

   c. Both sets are subranges of the same ______________.

CHAPTER TEN ACTIVITY

For each set operation, give the set that results:

Example:

	[1, 2, 3] + [4, 5, 6]	=	[1, 2, 3, 4, 5, 6]
1.	['a'..'z'] * ['a'..'f']	=	[ ]
2.	[monday, tuesday] - [monday]	=	[ ]
3.	[1..25] * [5, 7, 9]	=	[ ]

For each relational expression, give the evaluation:

Example:

	monday IN [monday, tuesday]	TRUE
1.	[1, 2, 3] = [2, 3, 4]	
2.	[1, 2, 3] <> [2, 3, 4]	
3.	[1, 2, 3] >= [2, 3]	
4.	[5, 6, 7] >= [3, 4, 5]	
5.	[1, 2] <= [0, 1, 2, 4, 8, 16]	

SECTION C: PASCAL EDITOR REFERENCE GUIDE

TABLE OF CONTENTS

PART ONE: THE NETWORK PASCAL EDITOR

The process of creating a Pascal program using Network Pascal involves four or five steps, as listed below. This guide concentrates on the first of these steps. The BEGINNER'S GUIDE (manual A) provided a demonstration of the entire sequence.

1. Access the editor program and enter your Pascal program.

2. Use the Network Pascal compiler (an option on the editor/compiler Main Menu) to compile the program (make it ready to run).

3. IF the compiler detected any errors, use the editor to correct these errors, and then recompile the program.

4. Move the source and object code onto a disk at the Host system. (Move is another option on the editor/compiler Main Menu.)

5. Exit the editor/compiler, and use the LINKLOAD or RUN program to run your Pascal program.

All editing occurs to the source file located in the memory of the Student Station. The user may move source and object files to and from the Host, but the Network Pascal editor itself does not use the network and cannot access files located on the Host.

1. HOW TO LOAD THE NETWORK PASCAL EDITOR

The Network Pascal editor is part of the NETPCL editor/compiler program provided on the Network Pascal diskette. To load the editor/compiler, type **NETPCL** and press **ENTER** at any Student Station that is displaying the **Network 3** prompt. The editor/compiler is then loaded, and a Main Menu is displayed.

Selecting **E** for "Edit" at the Main Menu gets you into the Network Pascal editor.

2. HOW TO ENTER A NEW PROGRAM

Unless a source program is currently stored in the Student Station's memory, you will see an almost blank screen when you get into the Network Pascal editor. The message **EOF** in the upper left corner of the screen marks the "end of file." A line of text at the bottom of the screen displays the number of program lines currently in the editor, and the number of bytes available (that is, the number of characters that there is room to add).

You enter a new program by entering Pascal program lines, using the special editing keys and features described on the following pages. Before you can type in a program line, you first need to add at least one line of blank space. Hold down the **SHIFT** key and press **@** to add one blank line, or to add a block of 16 blank lines press **CLEAR** followed by **E** (or Model 4 users can press the **F1** key to add a block of 16 blank lines). You can then begin adding text, using the special keys and editing features in compose mode and command mode as described below.

The screen can display up to 15 lines of the source file at any time. The source file in the Student Station's memory can of course contain more than 15 lines.

3. HOW TO EDIT AN EXISTING PROGRAM

Editing an existing program involves the same keys and features as entering a new program. If the program is not already in memory at the student station, select **M** at the Main Menu and move the program into memory. Use of the **M** option to "Move files" is discussed on page A-5 of the BEGINNER'S GUIDE.

4. THE SPECIAL EDITING KEYS AND FEATURES

The Network Pascal editor has two modes: compose mode and command mode. In compose mode, editing functions are accomplished by using simple, pre-defined key sequences. In command mode, editing functions are accomplished by entering the name of a command, sometimes followed by parameters for the command. Let's begin with a look at compose mode.

4.a. COMPOSE MODE

In compose mode, commands are available to move the cursor, delete text, insert text, set tabs and indents, modify text format, and search through program text. Descriptions of these commands follow. When **SHIFT** and a key are indicated in the following descriptions, **SHIFT** should be held down while the key is pressed. When **CLEAR** and a key are indicated, **CLEAR** should be pressed and then the key should be pressed. The Appendix to this EDITOR GUIDE lists these commands in brief, along with alternate ways of representing some of these commands, using **CTRL** and another key.

Cursor Movement

The cursor may be positioned anywhere on the screen by pressing any of the following labeled keys while in the compose mode.

↑	Moves cursor up one line. If the cursor is at the top of the screen, then the screen scrolls one line. If the cursor is at the top line of the file, then this key has no effect.
↓	Moves cursor down one line. If the cursor is at the bottom of the screen, then the screen scrolls one line. At the last line of the file, this key has no effect.
→	Moves cursor to the right one character. If the cursor is at the right edge of the screen, then no movement occurs.
←	Moves cursor to the left one character. If the cursor is at the left edge of the screen, then no movement occurs.
ENTER	Moves cursor to the beginning of the next line.
CLEAR ←	Moves cursor to the beginning of the current line.
CLEAR →	Moves cursor to the end of the current line.
CLEAR ↑	Scrolls the display one page toward the beginning of the file. (A "page" is fourteen lines unless you have used the **ROLL** command in "command" mode to otherwise define the "page.")
CLEAR ↓	Scrolls the display one page toward the end of the file.
CLEAR T	Moves the cursor one tab position to the right. If the cursor is at the right edge of the screen, then the cursor wraps around to the leftmost tab stop on the same line. (Tabs are set every three characters unless you have used **CLEAR S** to otherwise define tabs.)
CLEAR B	Moves the cursor one tab position to the left. If the cursor is at the left edge of the screen, then the cursor wraps around to the rightmost tab stop on the same line.
CLEAR H	Moves the cursor to the top left of the display ("home").

Text Deletion

SHIFT →	Deletes the character at current cursor position.
SHIFT ←	Deletes the entire current line.
CLEAR K	Deletes text from cursor to the end of the current line.

NOTE: A special feature of the Network Pascal editor is that the last line deleted with **SHIFT** and the left-arrow may be undeleted. If you press **CLEAR** followed by **U**, the last line that was deleted will be inserted into the text at the current cursor position. This feature may be used (with care) to move lines of text. Simply delete a line, move the cursor, and undelete it. Note that only one line is saved at a time.

Text Insertion

CLEAR I	Enters the "insert character" sub-mode. This allows text to be inserted anywhere in an existing text line. Steps: (1) place the cursor at the desired location, (2) press **CLEAR I**, and (3) type in the text. To exit this mode, press any cursor movement key.
SHIFT @	Inserts a blank line into the text buffer.
CLEAR D	Duplicates the line above the cursor onto the line the cursor is currently on. Text to the right of the cursor position is replaced by a copy of the text on the line above.

Compose Mode Commands for Editor Parameters and Settings

CLEAR A	Toggles the auto indent setting. If auto indent is OFF, **CLEAR A** turns it ON; if ON, **CLEAR A** turns it off. (Auto indent causes the **ENTER** key to align the cursor with the first non-blank character on the next line. If the next line is blank, the cursor is placed below the first non-blank character on the line above. This feature is useful when programming with indentation.)
CLEAR S	Sets a typewriter-like tab stop at the current cursor position. (The editor has preset tab stops every three characters which may be cleared in command mode using the **TABS** command.)

CLEAR Y	If a tab exists at the current cursor position, it is cleared.
CLEAR ?	Displays the amount of unused memory available in the editor's text buffer.
CLEAR C	Gets you into "command" mode.

Text Modification

CLEAR G	Appends the line after the cursor onto the end of the current line.
CLEAR O	Splits the current line in two at the cursor position.
CLEAR F	Searches forward in the text buffer for the next occurrence of a particular string. (The string to be searched for must be loaded into the "find string buffer" using the command mode **FIND** command.
CLEAR R	Searches forward in the text buffer for the next occurrence of the string in the "find string" buffer, and replaces this string with another string. The replacement string is specified in command mode using the **REPLACE** command.

Special Characters

Pascal characters that are not on the TRS-80 keyboard may be generated by pressing the **CLEAR** key followed by a digit. The entire list is shown below.

Key sequence	Character
CLEAR 0	~
CLEAR 1	[
CLEAR 2	]
CLEAR 3	^
CLEAR 4	{
CLEAR 5	}
CLEAR 6	|
CLEAR 7	_
CLEAR 8	\
CLEAR 9	`

4.b. COMMAND MODE

To get into command mode, press the **CLEAR** key followed by the letter **C** when you are in compose mode. Command mode is used to enter more complex commands than those used in compose mode, or commands that require parameters to be entered.

In command mode, angle brackets along with the cursor are displayed in the lower left corner of the editor screen. The user simply enters the name of an editor command. If you make a typing error in the command name, you can use the left-arrow key to back up and edit the command name before pressing **ENTER.** If you do not wish to complete the command you are typing, press **SHIFT ←** to return to compose mode.

Commands that have parameters can be entered in two ways. You can simply enter the parameters on the same line as the command. For example: **<>ROLL 10** tells the editor that you want the screen to scroll 10 lines at a time when **CLEAR ↑** or **CLEAR ↓** is used in compose mode. Alternately, you can enter the command name by itself and the editor will prompt for the parameters. If you enter **<>ROLL**, the editor will respond with: **<ROLL>LINES:.** At this point, you could type **1Ø** and press **ENTER.**

Commands available in Network Pascal's command mode are described below. When parameters are required, lowercase words describe the parameters.

FIND "string"

The **FIND** command will search forward in the text buffer (starting at the cursor position) for the specified string. If the string is found, the cursor will be positioned at the first occurrence, at the beginning of the string. If the string is delimited by quotation marks, then leading or trailing blanks will be included in the search string. (For subsequent serches for the same string, the compose mode's **CLEAR F** command may be used.)

Example: **FIND "INTEGER"**

REPLACE "old string" "new string"

REPLACE will search forward in the text buffer for the old string (starting at the cursor position). If the old string is found, then the first occurrence will be replaced with the new string, and the cursor will be repositioned at the beginning of the new string. (For subsequent searches and replaces using the same strings, the compose mode's **CLEAR R** command may be used.)

Example: **REPLACE "INTEGER" "REAL"**

QUOTE "string"

The **QUOTE** command is used when it is desirable to insert some non-printable characters into the file. It is also useful for inserting certain printable characters that are not on the TRS-80 keyboard into the file. The quoted string is inserted at the current cursor position. Non-printable characters may be represented by a # followed by a two character hexadecimal number (that character's ASCII representation). For example, **#5B** is the left bracket. Non-printable characters such as formfeed (**#ØC**) will not be displayed by the editor. However, the **FIND** command may be used to locate such characters in the buffer.

+ numberoflines   - numberoflines

The **+** and **-** commands are used to position the cursor a specified number of lines forward or backward relative to the current cursor position. (The blank after the **+** or **-** is required.)

ROLL numberoflines

ROLL will set the page size for all scrolling commands available in compose mode. The "page" is defined as 14 lines by default upon editor invocation.

SHOWLINE linenumber

The **SHOWLINE** command is useful for positioning the cursor absolutely to any line in the source file.

POSITION

Positions cursor on the screen at the position specified. Prompts that the user will see after entering this command are **<POSITION>ROW:** and **<POSITION>COL:**. The upper left corner of the screen is position **Ø,Ø.**

TABS integer

TABS = integer,integer,integer.....

TABS integer will set a tab stop every "integer" positions. All tabs may be cleared by setting integer to **Ø**. The editor defaults to a tab stop every third character position. (That is, tabs are set every three character positions unless you specify otherwise.) An alternate form of the command is **TABS = integer,integer,integer.....** This alternate form sets tabs at each specified column.

QUIT (answer yes or no)

QUIT will abort an edit session and delete the current source file in memory.

EXIT

EXIT ends an editing session and retains the current source file in memory. The Main Menu is redisplayed.

5. NOTE ABOUT EDITOR MEMORY USE

All of the text manipulated by the Network Pascal editor is in the source file in the student station's memory. The size of this file is limited by the amount of memory available. When the editor runs out of memory, it will display the message **MEMORY EXHAUSTED.**

One way to obtain more space is to delete the object file. Since the source and object files must share an area in memory, space that is used for one is not available to the other. It is possible that an object file exists in memory from a previous compilation of the program you are working on or from a previous program or user. To delete the object file, simply exit the editor and use the **D** option. You can then enter the editor again and continue editing.

The Network Pascal editor does not count blank spaces in your source code as used memory. At any time, you can find out how much memory is available by pressing **CLEAR** followed by the question mark **(?)**.

PART TWO: THE BLAISE EDITOR FOR MODEL III PASCAL

> NOTE: Before using TRS-80 Pascal with LDOS, turn to Appendix II of the MODEL III PASCAL SYSTEM REFERENCE GUIDE for information on patches you may need to make.

1. DISK AND FILE INFORMATION FOR USING THE MODEL III PASCAL EDITOR

The Blaise editor system for Pascal on a Model III or 4 stand-alone system is composed of the main editor file named **ED/CMD** and several help files labeled with the **/HLP** filename extension. These files are stored on Model III Pascal Disk One, and may be copied to any disk for use on a stand-alone Model III or 4 disk system.

The help files contain helpful messages that may be viewed while in the text editor. They are not required to be present on the same disk as the **ED/CMD** program; but if they are not, no help information may be obtained during the edit session.

File management using TRS-80 Pascal differs significantly from Network Pascal. The information that follows tells you what you need to know to effectively edit files using TRS-80 Pascal (on a stand-alone system).

1.a. EDITOR WORK FILE

When you edit a currently existing file, the text editor creates a new text file that is a copy of the file being edited. All editing changes are made to this work file, which has the temporary filename **T011/TMP.** This is a protection feature, designed to insure that the file is not destroyed through any hardware or software errors that may occur during the edit.

When the user executes an **EXIT "filename"** command (in command mode) to stop editing, the editor exits, consistency checks are made, and the work file becomes the new copy of the edited file. The old copy is then deleted. Completion of this process requires that there always be some free disk space available on the disk where the work file exists. If there is not sufficient free space during an exit, the disk on which the work file or the original file is placed will run out of disk space. In this case, the editor will flag an error message, allowing an abort or appropriate actions to be taken.

If an error occurs during the **RENAME** part of the exit command, (that is, while the work file is being copied back to the original file) then **T011/TMP** should contain the edited file. It may be used to restore the original file should the original become damaged. The work file is placed on the lowest numbered non-write-protected drive unless a file name with a drive specifier was used when invoking the editor. (In this case, the work file is placed on the specified drive.)

1.b. TYPES OF FILES THAT MAY BE EDITED

New files may be created or old files edited using **ED/CMD.** The size of files that may be edited is only limited by the disk space available on a disk. This allows creation of longer Pascal programs than is possible with Network Pascal, where limitations are based on the size of memory in Student Stations.

Files may not be split across disk boundaries. This means that the whole file must reside on a single disk. The file name syntax (legal structure of filenames) is identical with that of the operating system in use (TRSDOS, LDOS, etc.). The text editor files are compatible with: normal TRSDOS BASIC files, TRS-80 Pascal source and object files, or any other ASCII-formatted files that comply with TRSDOS file conventions.

1.c. REMOVAL OF DISKETTES FROM DRIVES DURING AN EDIT SESSION

The diskette containing the **ED/CMD** file may be removed after an edit session has begun, subject to the following restrictions:

1. There must always be a diskette with an operating system installed in the designated system drive. This diskette may be swapped during the edit session as long as the new diskette contains a valid operating system and the change does not violate conditions 2-4.

2. Removal must not cause the diskette containing the editor workfile to be removed.

3. If help files are removed, then no help messages will be available.

4. Before exiting the editor or appending lines to the text buffer, you must replace in the drive the diskette containing the original file.

NOTE: If the above rules are followed, you may change diskettes in order to use the editor's **INSFILE** command (to insert a portion of an existing file into the text buffer), as described on page C-14.

1.d. TEXT BUFFER MANAGEMENT

The editor maintains a fixed-size buffer for storing text. The buffer will hold approximately 13000 characters. All editor commands except for specific file commands operate only on the text in this buffer. When you are editing very large files, the file must be edited a section at a time. Starting at the beginning of the file, a section is loaded into the text buffer. Before another section of the file can be loaded into the buffer, buffer space must be made available by writing the text out to a work file. Then the next section may be loaded into the buffer. This process may be repeated until the whole file has been loaded and edited.

When editing an existing file, the editor loads the first 100 lines only. This leaves ample buffer space for adding more lines and performing the various editing functions. If the file is longer than 100 lines, you can use the **APPEND** command to load more text from the file into the buffer. With this command, you specify how many lines to copy from the file to the buffer. The copying begins one line past the last line previously loaded from the file. The text being copied from the file is appended to the end of the text in the buffer.

If the file is very large, it is possible for the buffer to become full. If this happens, a **MEMORY EXHAUSTED** message is displayed. The **WRITE** command must then be used to write some of the text in the buffer back out to a work file. With this command, you specify how many lines to copy from the buffer to the work file. The copying begins with the first line in the buffer and continues until either the buffer is empty or the specified number of lines have been written. Once lines have been written from the buffer to the work file, they may not be edited again during the current edit session.

If the editor is exited before the entire file has been loaded into the buffer, the editor will copy the remaining lines in the original file to the work file.

1.e. WORK FILE

If the editor work file is placed on the same diskette as the original file, then there must be enough space on the diskette at all times for two copies of the edited file.

The placement of the work file on a specific drive and diskette depends on how the editor is invoked. Under TRSDOS, if the filename drive specifier is not appended to the filename when the editor is loaded, the editor will search the various drives and place the work file on the lowest-numbered drive encountered that has enough space. If a drive specifier is appended to the filename when the editor is called, then the work file will be placed on the same diskette as the original file.

Under TRSDOS, if the file being edited is a new file (and no filename is specified when the editor is loaded), then the work file will be placed on the lowest-numbered drive that has enough space.

2. EDITOR COMMANDS FOR MODEL III PASCAL

This section assumes that you have already read PART ONE: THE NETWORK PASCAL EDITOR (pages 1 - 8 of this EDITOR GUIDE). The Pascal editor for stand-alone systems is quite similar to the Network Pascal editor. This section is devoted mainly to a discussion of differences between the two editors.

2.a. HOW TO ACCESS THE EDITOR

To create a new file, type **ED** when you see the **TRSDOS Ready** message, and press **ENTER.**

To edit an existing file, type **ED filename** at **TRSDOS Ready** (where **filename** is replaced by the name of the existing file), and press **ENTER.**

As with the Network Pascal editor, once the TRS-80 Pascal editor is loaded the screen will display either a blank area (for a new file) or the first lines of the file (for an existing file). At this point, the editor differs from the Network Pascal editor in two ways:

- Instead of the **EOF** (end of file) message, an **EOB** (end of buffer) message is displayed.
- A block of 16 blank lines cannot be added using Model III Pascal. Instead, blank lines are added one at a time when the user presses the **SHIFT @** key combination in compose mode.

2.b. COMPOSE MODE

Like the Network Pascal editor, **ED/CMD** has two modes: compose mode and command mode.

The compose mode editing keys and features that can be used with **ED/CMD** are identical to those used in the Network Pascal editor. See the Appendix to this EDITOR GUIDE for the complete list.

2.c. COMMAND MODE

As in the Network Pascal editor, command mode is entered using the **CLEAR C** key sequence in compose mode. Command mode is indicated by angle brackets and the cursor in the lower left corner of the screen. The command mode keys and features that can be used with **ED/CMD** are identical to those used in the Network Pascal editor with the following additions:

APPEND numberoflines

The **APPEND** command reads text from the original file and appends it to the end of the text buffer. After the append is executed, a message will appear at the bottom of the display with the total number of bytes available for additional text. If memory becomes exhausted, then text must be written to the work file using the **WRITE** command.

WRITE numberoflines

The **WRITE** command will write to the work file the specified number of lines, starting from the first line in the text buffer. As the write occurs, buffer space is released. Once the lines have been written to the work file, they may no longer be edited during the current session. They are permanently saved in the work file.

HELP topic

The **HELP** command will display helpful messages on the screen regarding the topic specified, if help for that topic is available. Supplied topics are: **HELP** (which provides general help information), **CMD** (which supplies command mode information), and **KEY** (which provides key definitions and compose mode information). If a topic is not specified, general help will be displayed. Help information may be viewed by using the same cursor movement keys that are active for the **SHOWFILE** command. To exit the help messages, press **CLEAR** followed by **C.**

SHOWFILE filename

The **SHOWFILE** command will open the specified file and show a portion of it. Several special commands may be issued while using **SHOWFILE.** They are:

CLEAR ↑	Scrolls the display up one page in the file.
CLEAR ↓	Scrolls the display down one page in the file.
linenumber	Positions absolutely to the line number in the file.
+ linenumber **- linenumber**	Scrolls the specified number of lines relative to the current cursor position. + rolls toward the end of the buffer; - rolls toward beginning of the buffer.
CLEAR C	Return to editing.

INSFILE "filename" startline numberoflines

This command will insert a portion of any pre-existing file into the text buffer starting at the current cursor location. **Startline** is the position in the file that is being inserted, where the insertion starts from. **Numberoflines** is the number of lines to insert. If the number of lines is greater than the last line in the file, then the insertion process stops at the last line. If the number of lines is greater than available memory in the buffer, then only the number of lines that will fit into memory will be inserted.

HSCROLL column

The **HSCROLL** command will scroll the display horizontally to the left or right. This feature allows editing of files wider than the TRS-80 screen. Once a horizontal scroll is performed, the display will remain in this mode until repositioned by another horizontal scroll command. The **column** parameter is the new column position for the left edge of the screen. Maximum column position that may be specified is 16.

EXIT "filename"

Unlike the Network Pascal editor's **EXIT** command, this command takes a filename as parameter. The **EXIT** command in this case writes out the entire text buffer to the work file. Any non-appended lines from the original file will also be written to the work file. Once consistency checks have been made, file renaming or deletions will occur. If you specified a filename when you entered the editor, you can respond with the **ENTER** key to the filename prompt. This will replace the old file that was edited. You can also specify the name of a new file. If the filename was left out when you called the editor, then a filename must be specified in the **EXIT** command.

3. DISK I/O ERROR DETECTION AND RECOVERY

When an error has been detected during diskette I/O operations, the message **IO ERROR** followed by the operation system error code will be displayed. You may type **Q** to allow the error to pass, or any other key to cause a retry.

APPENDIX I: SUMMARY OF EDITOR COMMANDS

CONTROL MODE

KEY	ALTERNATE	FUNCTION
↑	CTL/U	Cursor up
↓	CTL/J	Cursor down
←	CTL/H	Cursor left
→	CTL/R	Cursor right
CLEAR ↑	CTL/B	Roll backward
CLEAR ↓	CTL/A	Roll forward
CLEAR ←		Cursor to BOLN
CLEAR →		Cursor to EOLN
SHFT →	CTL/P	Delete character
SHFT ←	CTL/N	Delete line
SHFT @	CTL/O	Insert line
ENTER	CTL/M	New line
CLEAR A		Toggle auto indent
CLEAR B	CTL/T	Back tab
CLEAR C		Command mode
CLEAR D	CTL/D	Duplicate line
CLEAR F	CTL/C	Find next string
CLEAR G		Merge two lines
CLEAR H	CTL/L	Home
CLEAR I	CTL/Q	Insert char mode
CLEAR K	CTL/K	Delete to EOLN
CLEAR O		Open line at cursor
CLEAR R	CTL/Z	Replace next string
CLEAR S	CTL/W	Set tab
CLEAR T		Tab
CLEAR Y		Clear tab
CLEAR ?		Display memory

COMMAND MODE

APPEND	Add text to buffer
EXIT	Exit and save file
FIND	Find string
HELP	Display help
HSCROLL	Horizontal scroll
INSFILE	Insert file
QUIT	Abort changes
QUOTE	Insert literal string
REPLACE	Replace string
ROLL	Set roll
SHOWFILE	Display a file
SHOWLINE	Display a line

DEVICE NAMES

:L	Line printer
:C	Screen
:D	DUMMY FILE

APPENDIX II:

CREATING ALCOR PASCAL SOURCE FILES WITH SCRIPSIT

The Model III Pascal and Network Pascal compilers can have as their input files created by any editor which is capable of outputting an ASCII file. All that is necessary is that the file be in the proper format, and have the proper source file syntax for the version of Pascal to be used.

For instance, to use Model III SuperSCRIPSIT® to create/edit a file to be used on Network Pascal:

1. <O>pen the file using any valid filename.

2. Write and edit the Pascal program, making sure that the column conventions, syntax, and general form of Network Pascal are followed.

3. Press **CLEAR** Q to quit editing and save the file.

4. Use the **A** option to convert the file to ASCII. It is probably a good idea to use a 'standard' Pascal source file name for the ASCII file (that is, a filename ending in **/PCL**).

5. If you are using a floppy host for your network, then **COPY** the file to a disk to be used on the network. If you have a hard disk-equipped host, then **CONV**ert it.

6. At the Student Station, load **NETPCL**, load the **MOVE FILES** overlay, load the **ASCII/PCL** file, then load the compiler overlay and compile the program.

Alternatively, you could have compiled the program (created on Super-SCRIPSIT) using Model III (stand-alone) Pascal, and then **COPY**ed or **CONV**erted the object file to the network to be later **RUN** or **LINKLOAD**ed.

SECTION D: LANGUAGE REFERENCE GUIDE

TABLE OF CONTENTS

INTRODUCTION

Organization of the Network Pascal Language Reference Guide

This reference guide assumes that the reader is already somewhat familiar with the Pascal language. It is organized as a reference source, with topics grouped in such a way as to make them easy to find. The guide was not intended to follow a progression of discussion that is well suited as a teacher of the Pascal language. If this is your first experience with the Pascal language, you should first work through the Pascal Tutorial (section B). The tutorial provides an introduction to the fundamental concepts and terminology of Pascal, and will give you the background you need to use this reference guide effectively.

Notation and Terminology used in this Reference Guide

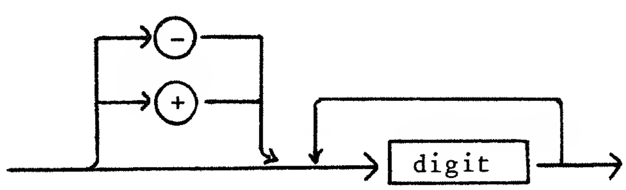
Both the syntax and the semantics of the programming language Pascal are described in this reference guide. Syntax refers to the arrangement of program elements into a form which the compiler can understand. Semantics refers to the meaning that the compiler associates with a particular arrangement of the program elements. The semantics of a language can be explained with words, but the syntax is best explained through the use of diagrams.

The syntax diagrams used throughout this reference guide describe the legal (allowed) syntax of program elements. Several legal forms are shown for most elements.

Each syntax diagram has an entering and an exiting point which is denoted by an arrow. The paths followed by these arrows represent the various different forms that the program element can take.

Let's consider an example. The following syntax diagram describes the syntax of an integer:

Syntax of an integer:

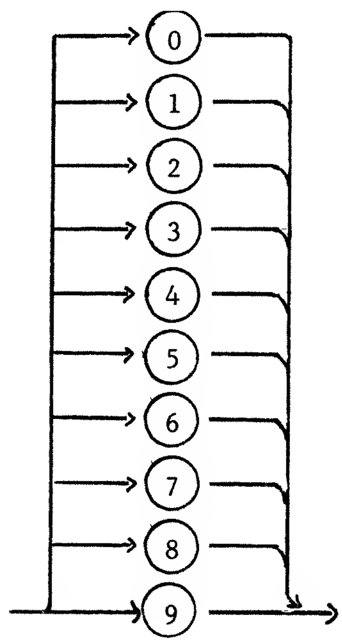


This diagram indicates that an integer consists of one or more digits optionally preceded by a plus or minus sign. Upon entering the diagram, you can select one of three possible paths. One path leads directly to **digit**, one leads to **+**, and one leads to **-**. The paths from **+** and **-** then lead to **digit.**

When you have passed through **digit**, you have so far constructed a one-digit integer optionally preceded by a positive or negative sign (for example, 2, -2, or +2). You now have the option of either exiting the diagram or following the arrow which leads back to the beginning of **digit** to add another digit (for example: 25, -25, +25). From this point, you may pass through **digit** repeatedly, or may exit the process after any pass. Thus, the diagram shows us that an integer may consist of one or more digits, optionally preceded by a positive or negative sign.

Let's consider another example. This second syntax diagram describes the ten correct forms that a digit may take. Entering the diagram, you have ten possible paths from which to choose. Each path leads to a single character that is a legal digit. You choose a path, follow it to the character and then exit the diagram.

Syntax of a digit:



In the syntax diagrams used in this reference guide, upper-case character strings denote reserved words (words with a set meaning in the Pascal language). These words must be present in the form shown. Lower-case character strings denote the parts of the syntax where many legal forms exist. For example, **integer** in a diagram represents any legal integer, while **INTEGER** represents a reserved word which must appear as shown.

In some cases, abbreviations are used to shorten a diagram. For example, **id** is used in place of **identifier**, and **expr** is used in place of **expression**. Where other abbreviations are used, their meaning should be apparent from the surrounding text.

CHAPTER ONE: PROGRAM ELEMENTS

The elements of TRS-80 Pascal (identifiers, numbers, strings, reserved words, and special symbols) are composed from the ASCII character set, listed in the appendix.

1. IDENTIFIERS

An identifier is a name that identifies a program, a constant, a type, a variable, a procedure, or a function. It consists of a letter followed by zero or more of the following characters:

- any of the 26 letters of the alphabet in lower or upper case
- any of the digits Ø through 9
- the character $
- the character _

Pascal makes no distinction between upper- and lower-case letters. The two identifiers **Name** and **name** are considered identical.

The identifier may be of any length, but only the first eight characters are significant to the compiler. Therefore, it is important to keep the first eight characters of identifiers unique. (For example, the compiler would read the identifiers **A2345678** and **A23456789** as identical, because all characters past the eighth character are ignored.)

An identifier cannot contain blank spaces or span a line boundary.

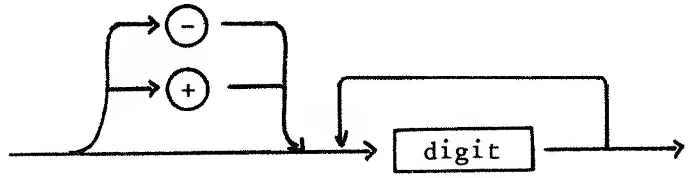
Examples of legal identifiers are:

Factor$ **DEPARTMENT**   **A**   **Div_1Ø**   **B12345678$_**   **C$123**

2. NUMBERS

Numbers are integer or real constants. The allowed range for an integer is -32768 to +32767. This limitation is imposed because an integer is allocated sixteen bits of storage. (You will remember that an integer may not contain a fractional part.)

Syntax of an integer number:

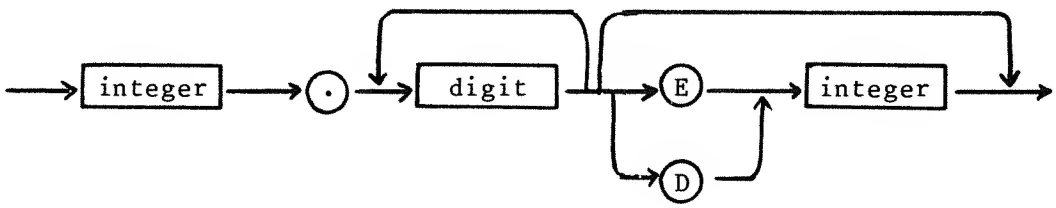


Examples of integer numbers: **30** **-28934** **0** **32739**

Real numbers are represented in either exponential or fixed-point form. The fixed-point form consists of an integer part followed by a decimal point and a fractional part (for example: **25.567**). The exponential form consists of a fixed-point part followed by an exponent part which is a multiplier (for example: **2.567E1**). The value of a real number expressed in exponential form is:

fixed-point part * (10 raised to the exponent part)

Syntax of real numbers:



Examples of real numbers:

Fixed-point form: **50.0** **-10000.0** **345.22452**

Exponential form: **0.239E3** **-4.5921E-2** **876.0E+33** **193.27E-3**

0.239E3 is equivalent to **239.0**
-4.5921E-2 is equivalent to **-0.045921**

NOTES:

The allowed range for real numbers is (-)1.7E-38 to (-)1.7E+38.

Using **D** instead of **E** in exponential form represents a double-precision real number. Single-precision real numbers are accurate to 6 digits; double-precision real numbers are accurate to 16 digits.

3. STRINGS

Strings are sequences of characters enclosed by single quotation marks. A string consisting of a single character is a constant of the type **CHAR**. Examples of legal strings are:

'ABC' **'12"QZW'** **'BEGIN'** **'***'** **' %'**

Strings consisting of **n** characters, where **n** is a number greater than one, are constants of the type **ARRAY[1..n] OF CHAR**. (The array is a structured data type discussed in Chapter Four of this reference guide.)

If a string is to contain a single quotation mark, the mark must appear twice in the sequence. The string consisting of the single character ' is represented as: ''''.

Characters in strings can optionally be denoted by a pair of hexadecimal numbers (base sixteen numbers composed of the characters 0 through 9 and A through F). The character # followed by 2 hexadecimal characters represents a single character in the ASCII character set. (See the ASCII character set in the Appendix to this guide for a key to what hexadecimal number pairs represent what characters.) One reason for using hexadecimal numbers is that they provide a mechanism for representing nonprintable characters.

To include the literal character # in a string, you must include it twice in the string; just as the character ' must appear twice when the character itself is to be made a part of the string. A string consisting of the single character # is thus represented as: '##'.

Examples of hexadecimal character representation in strings:

'#30' is equivalent to the letter '0'
'D#4FG' is equivalent to 'DOG'
'#44#4F#47" is also equivalent to 'DOG'
'#00' corresponds to the nonprintable null character
'A#B' is illegal

4. RESERVED WORDS

The following words are Pascal keywords and have special meanings in a program. They may not be used as identifiers:

AND	DOWNTO	IF	OR	THEN
ARRAY	ELSE	IN	PACKED	TO
BEGIN	END	LABEL	PROCEDURE	TYPE
CASE	FILE	MOD	PROGRAM	UNTIL
CONST	FOR	NIL	RECORD	VAR
DIV	FUNCTION	NOT	REPEAT	WHILE
DO	GOTO	OF	SET	WITH

5. SPECIAL SYMBOLS

The special characters below are used as operators and delimiters in a program. Because character sets vary from system to system, alternate representations are provided for some of the symbols.

Symbols with only one representation:

+	-	*	/		
=	<>	<	<=	>=	>
(	)	'	:=	.	,
;	:	#	::		

Symbols with alternate representations:

symbol	alternate
{	(*
}	*)
^	@
[	(.
]	.)

The following symbols can be produced by the TRS-80 Models III and 4 by using particular key combinations.

symbol	key combination
[	**CLEAR 1**
]	**CLEAR 2**
^	**CLEAR 3**
{	**CLEAR 4**
}	**CLEAR 5**
|	**CLEAR 6**
_	**CLEAR 7**

In addition, these two symbols can be produced by Network Pascal:

symbol	key combination
\	CLEAR 8
`	CLEAR 9

6. COMMENTS

Comments can be used in a program for documentation purposes. The compiler generates no code for comments.

The symbol { or (* denotes the beginning of a comment. The symbol } or *) denotes the end. All characters in between are ignored by the compiler.

Examples of legal comments:

```
{This is a comment.}

(*This is a comment
              that spans more than one line.*)
```

Comments may not be nested. The following will generate an error:

```
(*outer (*inner level*) level*)
```

7. THE SEMICOLON

The semicolon is used extensively in the Pascal language to separate the individual components of a program. For example, block headings must be separated from block parts, block parts must be separated from one another, and individual definitions, declarations, and statements within the block parts must be separated.

In general, the semicolon may be used freely throughout the program. However, care should be taken not to include a semicolon in the middle of a statement. This is a common source of error when using the **IF** statement with one or more **ELSE** clauses. [**IF . . . ELSE** is a conditional statement, discussed in Chapter Eight of this guide.] Since the **ELSE** clauses are a part of the **IF** statement, they must not be separated from it by a semicolon. An **ELSE** keyword should never be preceded by a semicolon.

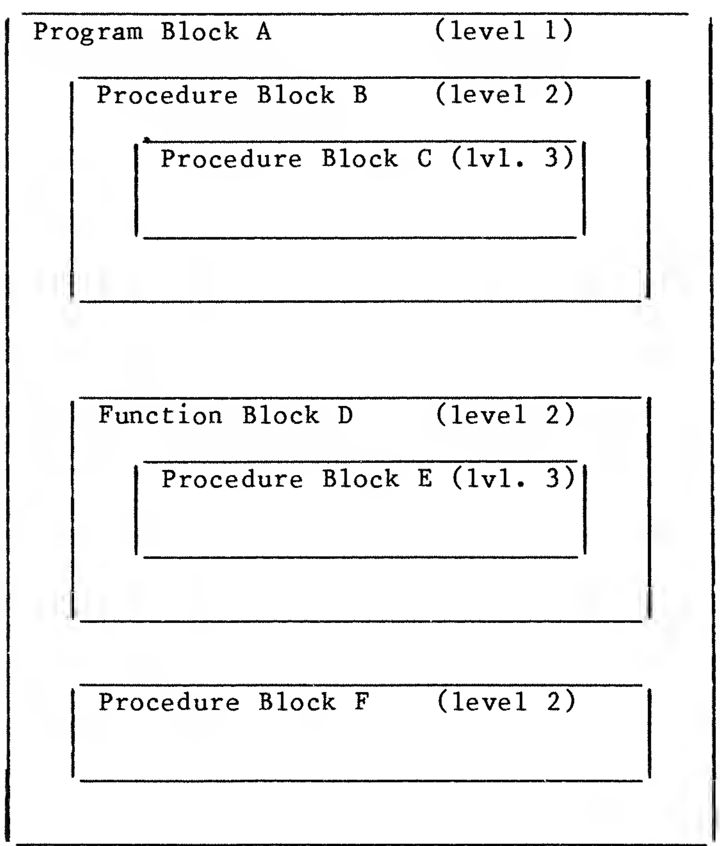
This example shows correct use of semicolons in an **IF** statement:

```
IF time > 12 THEN
    BEGIN
    alpha := 'e';
    beta  := 'f';
    END      (*semicolon here would cause an error*)
ELSE
    BEGIN
    alpha := 'g';
    beta  := 'h';
    END;
```

Chapter One has covered the fundamental elements of TRS-80 Pascal: identifiers, numbers, strings, reserved words, special symbols, and the semicolon. Chapter Two goes on to introduce Pascal program structure.

Pascal is a block-structured language in that a Pascal program is constructed out of units that can be compared to building blocks. A Pascal program consists of a minimum of one block. Additional sub-blocks (procedures and functions) may be created and placed within the "program block". The term for this process is "nesting." (Procedures and functions were covered in the Tutorial and are reviewed in Chapter Nine of this guide.)

The rule for nesting is that a block may lie entirely within another block, but blocks do not overlap in any other way. A level of nesting can be assigned to each block of a program. The block-structured organization of a program is represented pictorially by the following example:



A program, then, consists of at least one block, the program block. Optionally it contains procedure and/or function blocks which are nested within.

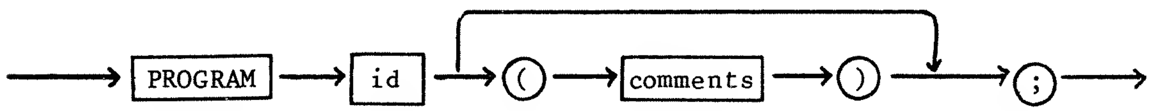
1. BLOCK HEADINGS

The purpose of the block heading is to give the block a name and, in the case of procedure or function blocks, to define any parameters (values) to be passed to the block from the main program. There are three types of blocks: the program block, the procedure block, and the function block. A single program has only one program block, the outermost block of the program. There may be any number of procedure and function blocks. Each of the three types of blocks has a different kind of heading. Let's look at these three kinds of headings.

1.a. The Program Heading

The program heading must be the first non-comment in a program. Its purpose is to signal the start of the program and to give the program a name. The heading consists of the word **PROGRAM** and an identifier (which is constructed according to the rules on page D-3) and a semicolon. Optionally, it may include comments enclosed by parentheses right before the semicolon. (Characters inside the parentheses are ignored by the compiler.)

Syntax of the program heading:



Example program headings:

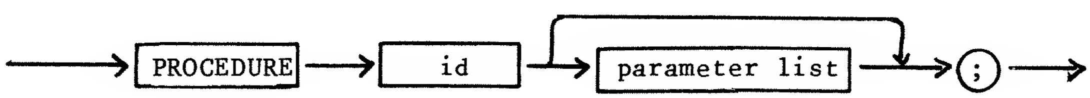
PROGRAM demonstrate;

PROGRAM taxes (computes income tax);

1.b. The Procedure Heading

The procedure heading signals the start of a procedure block. It gives the procedure a name and defines the parameters to be passed to it.

Syntax of the procedure heading:



The parameter list declares the variables which are used to pass data into and out of a procedure. Variables named in the parameter list are called formal parameters.

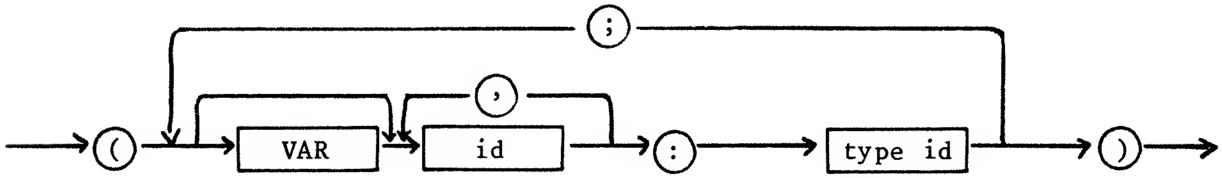
A statement outside of the procedure activates (or calls) the procedure. This statement has a corresponding list of parameters which are called actual parameters. The actual parameters must match the formal parameters in order and in type. However, their names need not be the same.

There are two kinds of formal parameters: pass-by-value or pass-by-reference. A pass-by-value parameter causes its corresponding actual parameter to be copied to another location and then the formal parameter references the copied value. Therefore, changing the value of the formal parameter inside the procedure does not change the value of the corresponding actual parameter.

In contrast, what is passed to a formal pass-by-reference parameter is the address of the corresponding actual parameter. The formal parameter references the same location as the actual parameter. Therefore, changing the value of the formal parameter also causes the value of the actual parameter to be changed.

Variable declarations in the parameter list which are preceded by the keyword **VAR** indicate pass-by-reference parameters. The absence of the keyword indicates a parameter that is passed by value.

Syntax of the parameter list is:



Example procedure headings:

PROCEDURE out;

PROCEDURE cpu(pc : INTEGER);

PROCEDURE delete(VAR i,j : INTEGER; ch : CHAR; VAR x : REAL);

In procedure **delete** above, **i** and **j** are integers which are passed by reference, **ch** is a character which is passed by value, and **x** is a real number which is passed by reference.

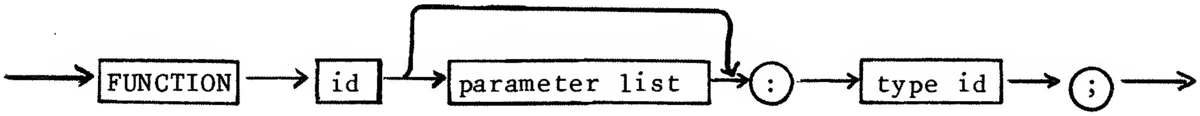
As a general rule, pass-by-value parameters should be used to prevent unpredictable side effects. However, side effects are sometimes desirable. That is, sometimes you may want a change in the value of a formal parameter to also change the value of its corresponding actual parameter. In such a case, pass-by-reference parameters should be used. Also, when you are passing large data structures such as arrays, pass by reference should be used. This speeds execution and saves memory because the program passes a pointer (address reference) that points to the structure rather than copying the whole structure to another location.

1.c. The Function Heading

The function heading signals the start of a function block. It gives the function a name and defines the parameters to be passed to the function.

Unlike a procedure, a function has a type associated with it, and, like a variable, has a value assigned to it. A function is referenced by an expression and its value is then substituted into the expression.

Syntax of the function heading:



The parameter list for a function has the same form as the parameter list for a procedure, discussed on the previous page.

Example function headings:

FUNCTION number : REAL;

FUNCTION nextstate(currentstate : INTEGER) : INTEGER;

In the example headings above, the function **number** is declared as a REAL-valued function which has no parameters. The function **nextstate** is an INTEGER-valued function which has one parameter, also of type **INTEGER.** Within each function, there should be an assignment statement which assigns a value to the function. For example, **number:=5.3** and **nextstate:=currentstate + 1** could appear inside the respective functions.

Note that a function may be of an ordinal type or of the type **REAL** only. A type is ordinal if the values which it can take form a finite set whose members can be placed in a linear order. The predefined ordinal types are **INTEGER**, **CHAR**, and **BOOLEAN.** You can define a new ordinal type by enumerating all of the values which the type can take.

2. Block Parts

A block may be composed of the following elements:

1. label declarations
2. constant definitions
3. type definitions
4. variable declarations
5. common declarations
6. access declarations
7. procedure and function declarations
8. the statement body

A block does not have to include all eight parts. At a minimum, a block must include the two keywords **BEGIN** and **END** which bracket the statement body. The following is an example of a minimum complete program. It contains only the program block, which is composed of only the heading and a null statement body.

```
PROGRAM donothing;
     BEGIN          (*The statement body contains no statements.*)
     END.
```

The eight parts listed are described in detail below. The first six of these parts may appear in any order. The only requirement is that an identifier must be defined before it is used. For example, the definition of a user-defined type must textually precede the declaration of a variable of that type. (The only exception to this is the definition of pointer types which are discussed in Chapter Five.)

There may be more than one of any particular "part." For example, there could be two separate type-definition parts.

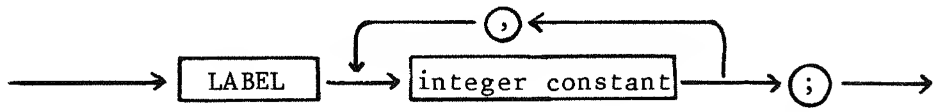
The procedure and function declarations follow any use of the first six parts. The statement body then follows the last procedure or function declaration.

2.a. Label Declarations

Label declarations are used in conjunction with the **GOTO** statement, which transfers program control to the destination named in its argument. A label declaration provides a means of labeling that destination. A **GOTO** statement can then reference the label, causing a branch to the statement.

The label declaration part is signaled by the keyword **LABEL.**

Syntax of the label declaration part:



A label must be declared in the same block where the **GOTO** statement that references it appears. Branching outside a block is not allowed. Several labels may be declared on a single label declaration line. All declared labels must appear somewhere in the statement body.

Example label declaration part:

LABEL 100, 200, 300, 400, 500, 1000;

Syntax used to label the statement:

Examples of labeled statements:

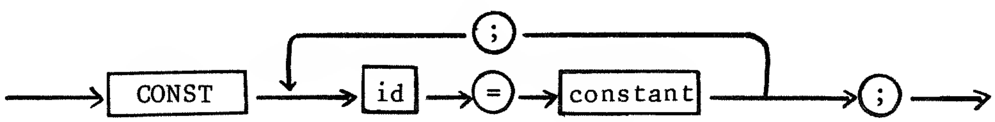
```
100: x:=47;

200: IF x > 500 THEN GOTO 100;
```

2.b. Constant Definitions

A constant definition is used to associate an identifier with a value which does not change. A constant identifier is assigned a value at compile time and this value cannot be changed without the program being recompiled. This means that a constant identifier cannot have its value changed by an assignment statement. The use of constant identifiers increases program readability because meaningful names (such as **pi**) can be used in the place of actual values (such as **3.14159265**).

The values which can be assigned to constant identifiers are numbers, strings, or other identifiers which are constants. This includes identifiers which are members of an enumeration (an enumeration is a list of the values which a particular variable or type can take on). The start of the constant definition "part" is signaled by the keyword **CONST**.

Syntax of the constant definition part:



Example constant definition part:

```
CONST
     low=32;
     high=88;
     pi=3.14159;
     speedoflight=299792.0;
     separator='---------';
     positive=10;
     negative=-positive;
     keydefinition=#61;
```

NOTES:

Integer constants may optionally be expressed using hexadecimal numbers preceded by #.

There is a predefined constant **MAXINT** which is defined to be equal to the largest positive value an integer can take.

1.c. Type Definitions

A type definition is used to create a new data type. (Some data types are predefined by Pascal; others are user-defined, new data types created by the Pascal programmer.) A type definition associates an identifier with a user-defined simple or structured data type. The identifier can then be used in a variable declaration to specify the type of the variable. For example, the identifier **colors** might be associated with a data type composed of the values **red, blue, green, orange, purple.**

A variable of a user-defined type can declare its type directly in the variable declaration part. For example:

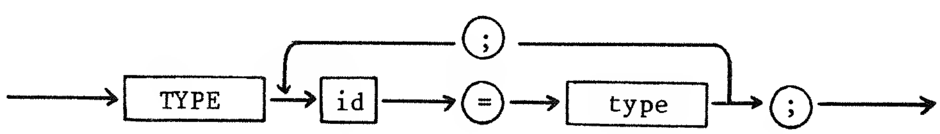
```
VAR
months  : (jan,feb,mar,apr,may,jun,jul,
           aug,sep,oct,nov,dec);
```

Even so, it is useful to have a name associated with a user-defined type, particularly when you are using structured types whose definitions are long, and when more than one variable in the program is to be declared of that type. Associating a name with the type means that the type must be defined only once.

In some cases, type definitions are necessary. If comparisons are to be made between two variables of a user-defined type, then the variables must be declared as the same type. Defining the type for each variable separately in a variable declaration part will not work. Although the variable declarations will look the same, the compiler will view them as variables of two separate types. Also, declarations of variables in the parameter list of a procedure or function must be to named types. For example, if an array is to be passed as a parameter, the array must be defined in a type definition and the formal parameter must then be declared as that type.

The type definition "part" is signaled by the keyword **TYPE.**

Syntax of the type definition part:



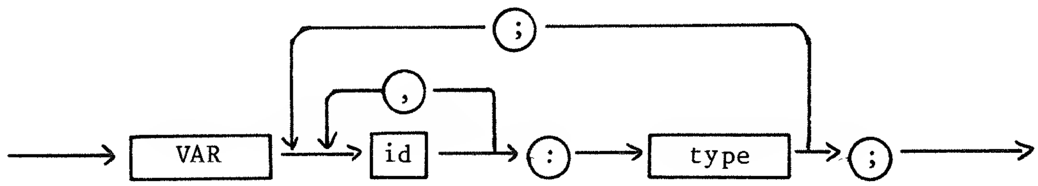
Example type definition part:

```
TYPE        colors = (red,blue,green,orange,purple);
            weekdays = (sunday,monday,tuesday,wednesday,
                        thursday,friday,saturday);
            workdays = monday..friday;
            daysofmonth = 1..31;
            letters = 'A'..'Z';
```

2.d. Variable Declarations

Each variable in a program must be declared before it is used. This is done by associating the variable name with a type. The type can be (1) the name of a predefined simple data type, (2) the name of a user defined type which has previously been defined in a type definition part, or (3) the type can be defined when the variable is declared. The start of the variable declaration part is signaled with the keyword **VAR.**

Syntax of the variable declaration part:



Example variable declaration part:

```
VAR
    x,y,z        : REAL;
    ok           : BOOLEAN;
    i,j,k        : INTEGER;
    fruit        : colors;
    alpha,beta   : CHAR;
    characters   : list;
    mark         : ARRAY [1..30] OF INTEGER;
    byte         : 0..255;
    months       : (jan,feb,mar,apr,may,jun,jul,
                        aug,sep,oct,nov,dec);
    account      : RECORD
                   number,date : INTEGER;
                   END;
```

In the example above, **x,y,z : REAL;** declares three variables to be of a predefined simple data type; **fruit : colors;** declares a variable to be of a user-defined type that has presumably been defined in the type declaration part; and the type of the variable **months** is defined in the variable declaration, by an enumeration of the values that **months** may take on.

2.e. Common Declarations (Available in Model III Pascal but not in Network Pascal.)

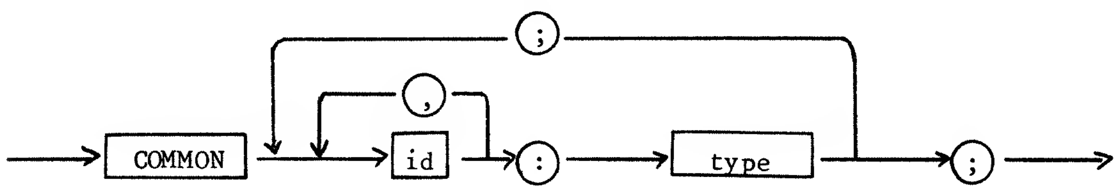
The common declaration part is used in the same manner as the variable declaration part to associate a variable name with a data type. However, declaring a variable in the common declaration part gives it a special property. Normally, storage space for variables is allocated dynamically. That is, variables declared in a block are allocated storage space when the block is activated, and the space is freed when the block terminates. Thus, the local variables of a block become undefined when the block terminates. In contrast, common variables are allocated storage space statically at compile time. This means that the common variables of a block retain their defined values even after the block terminates.

Common variables are "scoped" similar to normal variables (see Chapter Nine of this guide for a discussion of scoping rules). However, only one storage location is reserved for each common variable name. Therefore, a common variable declared locally within a procedure will reference the same location as a common variable of the same name declared anywhere else in the program.

A common variable cannot be accessed in a block unless its name appears in an access declaration of the same block. This feature is useful for controlling access to global variables, providing protection and better documentation of where global variables are used.

Another valuable use for common variables is in external procedures. A procedure which is often used by many separate programs can be compiled separately and linked to the programs that use it. In the case where the procedure must retain information between activations, such as information on the cursor position in a graphics procedure, common variables may be used to prevent the need for global variables.

Syntax of the common declaration part:



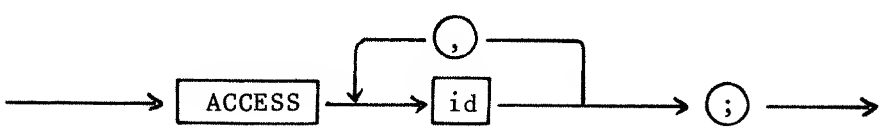
Example common declaration part:

COMMON cursorx,cursory : INTEGER;

2.f. Access Declarations

An access declaration is used in conjunction with a common variable. No common variable can be referenced unless its name appears in an access declaration of the block which references it. The order in which common variable names appear in an access declaration is arbitrary.

Syntax of the access declaration part:



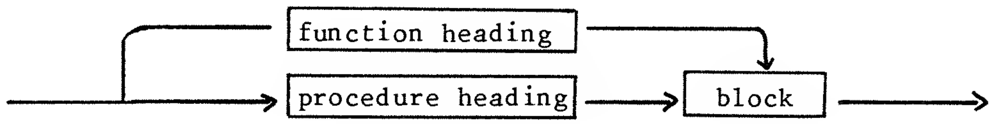
Example access declaration part:

ACCESS cursorx, cursory;

2.g. Procedure and Function Declarations

A procedure or function declaration creates a new block. A procedure declaration consists of a procedure heading followed by a block. A function declaration consists of a function heading followed by a block. Procedure and function declarations form subblocks within the block in which they appear.

Syntax of procedure or function declaration:



Example procedure declaration:

```
PROCEDURE getvalue(first,last :INTEGER; VAR word : buffer;
                                        VAR value: INTEGER);
(*Converts hex character string to decimal value:
  buffer is a globally declared type --> PACKED ARRAY[1..8] OF CHAR;
  word contains the hex character string
  first and last are pointers into the string
  value is the returned decimal value                              *)
VAR  i,n,factor     : INTEGER;
     ch             : CHAR;
BEGIN
  value := 0; factor := 1;
  FOR i := last DOWNTO first DO
     BEGIN
     ch := word[i];
     IF ch = ' ' THEN n:=0    (*Blank character given value 0*)
     ELSE
        IF (ch>='0') AND (ch<='9') THEN   (*character range 0..9 *)
           n := ORD(ch)-ORD('0')          (*convert ch to decimal*)
        ELSE
           IF (ch>='A') AND (ch<='F') THEN  (*character range A..F *)
              n := ORD(ch) - ORD('A') + 10; (*convert ch to decimal*)
     value := value + factor * n;
     factor:= 16 * factor;            (*hex is base 16*)
     END;
END;                                  (*procedure getvalue*)
```

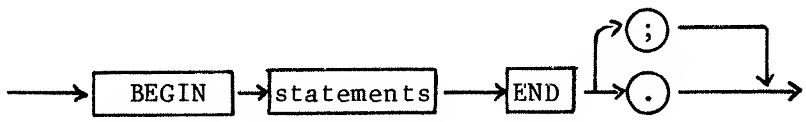
Example function declaration:

```
FUNCTION nextstate(currentstate : INTEGER) : INTEGER;
(* returns the next state given the current state *)
BEGIN
     CASE currentstate OF
        1: nextstate := 3;
        2: nextstate := 4;
        3: nextstate := 1;
        4: nextstate := 2;
     END;
END;                                 (*function nextstate*)
```

2.h. Statement Body

The statement body of a block contains zero or more statements which describe the actions of the block. The statement body must start with the keyword **BEGIN** and end with the keyword **END**. However, since individual statements may also include **BEGIN** and **END**, the statement body may contain many occurrences of these two keywords. The statement bodies for the three types of blocks are identical, except that the concluding **END** for the program block statement body must be followed by a period. The concluding **END** for procedure and function statement bodies must be followed by a semicolon.

Syntax of statement body:



Example statement body:

```
BEGIN          (* begin program block statement body *)
     WHILE NOT EOF DO
          BEGIN
          READ (x,y,z);
          X :=SQR(x); y := SQR(y); z := SQR(z);
          WRITE( 'squaredata ' , x , y , z);
          END;
END.            (* end of program *)
```

Next, Chapter Three discusses the simple data types that are predefined in Pascal or may be defined by the user.

CHAPTER THREE: SIMPLE DATA TYPES

The "simple" data types consist of ordinal types and the type **REAL.** They form the base for building structured types.

1. ORDINAL TYPES

A type is said to be ordinal if it is characterized by a linear ordered set of distinct values which can be mapped on the set of natural numbers. This mapping is actually an enumeration (listing) of all the values which the type can take. The predefined ordinal types are **INTEGER**, **CHAR**, and **BOOLEAN.** New ordinal types can be defined by enumerating all the values which the type can take. In addition, new ordinal types may be defined as subranges of other ordinal types.

1.a. The Type INTEGER

Variables declared as type **INTEGER** may take on values in the range -32768 to +32767. All of the arithmetic operators (example: + and -) and the relational operators (example: > and <) can be used with integer constants and variables. [Note: The relational operator **IN** is used with integers only in conjunction with sets of up to 256 non-negative members (see Chapter Four).]

Integer calculations which cause an overflow will not generate an overflow error. For example, adding one to the predefined constant **MAXINT** (defined as the maximum value an integer can take on) causes a "wrap-around" to the lowest allowed integer value, -32768.

Syntax of type integer:

$$\longrightarrow \boxed{\text{INTEGER}} \longrightarrow$$

Example declaration:

```
VAR        i,j,k : INTEGER;
```

Example integer constants:

59   -1   0   329   -10000   29872

1.b. The Type CHAR

Variables declared as type **CHAR** can take single characters as values. The set of valid single characters is the ASCII character set. Each character has an associated ordinal number in the range 0 to 255. (A table of ASCII characters with associated ordinal numbers is listed in the Appendix to this guide.)

There are two functions which may be used with the character set. The function **ORD('character')** returns the ordinal number of the character specified. The function **CHAR(ordinal number)** returns the character associated with the ordinal number specified. For example:

ORD('A') would return the number **65**

CHAR(65) would return the character **A.**

These are called "transfer functions" because they are used to transfer a character value to an integer value and vice versa. Constants of the type **CHAR** are denoted by using single-character strings. All relational operators may be used with variables and constants of type **CHAR.**

Syntax of the type **CHAR**:

Example declaration:

VAR alpha,beta : CHAR;

Example character constants: **'9'** **'a'** **'#9F'**

Example relational expression: **'A' < 'B'**

1.c. The Type BOOLEAN

The **BOOLEAN** type represents logical values, represented by the predefined identifiers **FALSE** and **TRUE.** These are the only possible values of a **BOOLEAN** variable or expression.

The boolean operators **AND**, **OR**, and **NOT** take boolean operands (expressions that can be evaluated as true or false) and yield boolean results. The relational operators listed below all yield boolean results:

= **<** **>** **<=** **<>** **>=** **IN**

An example of a boolean expression is:

NOT alpha < beta AND gamma = 5

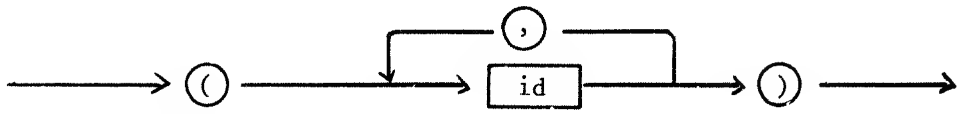
Example declaration: **VAR switch : BOOLEAN;**

Boolean constants: **TRUE** **FALSE**

1.d. The Enumerated Type

Pascal allows you to define your own ordinal types. You may create a new type by giving the new type a name and enumerating all the values that the type may take.

Syntax of the enumerated type:



Examples of enumerated types:

names = (Fred, Joe, Nancy, Susan);

foods = (hotdog, hamburger);

The values listed are identifiers. The order in which the identifiers are listed defines a relationship. The identifiers can be thought of as being mapped on to a set of natural numbers. The first identifier maps to 0, the second to one, the third to two, and so on.

This implies that:

identifier1 < identifier2 < identifier3...< identifierN.

For example, consider the predefined type:

BOOLEAN = (FALSE,TRUE);

The boolean value FALSE is less than the boolean value TRUE because FALSE appears in the list before TRUE. This kind of ordered relationship applies to any enumerated type. Consider the type definition:

colors = (red, blue, green);

By this definition, a variable declared as type **colors** can take on the value **red**, **blue**, or **green.** The definition also implies that **red < blue < green.**

This ordering means that enumerated values can be used in relational expressions. It also means that they may be used for range specifications. For example, consider the **FOR** statement. (**FOR**, introduced in Chapter Eight, is used to direct the computer to repeat a series of actions -- that is, to loop -- until a particular variable reaches a particular value.) The range of the loop-control variable is defined by specifying a starting and stopping value. These starting and stopping values could be the values of an enumerated type. For example, if a variable **color** has been declared as type **colors**, the following statement is valid:

FOR color := red to green DO

1.e. Subrange Types

A subrange type is simply a type defined to take on a subset of the values representing some ordinal type.

Syntax of the subrange type:

The use of subranges can sometimes save memory. For example, an integer variable whose values are always in the range of 0 to 255 could be declared as a subrange of the type **INTEGER.** You might define a new type as follows:

```
    byte  = 0..255;
```

Now, variables declared as type **byte** would be allocated 8 bits of storage rather than the 16 bits which are allocated for variables declared as type **INTEGER.** (This is because the digits 0 through 255 can be expressed in the computer's base-two number system using only eight binary digits.) The compiler allocates the minimum amount of storage required to represent the range of values specified by a subrange type.

Use of subranges can contribute to good documentation because they define the range of legal values that a variable declared as the subrange type can take on. Subrange types are also often used in conjunction with **SET** types which are discussed in Chapter Four.

2. THE TYPE REAL

The type **REAL** is used to represent fractional numerical data. The implementation of real numbers is dependent on the computer you are using. For the TRS-80 Model III or Model 4, the allowed range is (-)1.7E-38 through (-)1.7E+38, where "E" indicates exponentiation. (That is, the minimum and maximum ranges are (-)1.7 to the power of + or - 38.)

Real numbers are either single-precision (accurate to 6 digits) or double-precision (accurate to 16 digits). See Chapter One of this language reference guide for more information on real numbers.

Syntax of type **REAL**:

Example Declaration: **VAR** **x , y , z** **:REAL;**

Example real constants: **2.3** **-129.345** **5.496E-14** **-7983.851D+23**

Chapter Three has provided a brief introduction to simple ordinal types. More complex, <u>structured</u> data types are covered in Chapter Four.

CHAPTER FOUR: STRUCTURED DATA TYPES

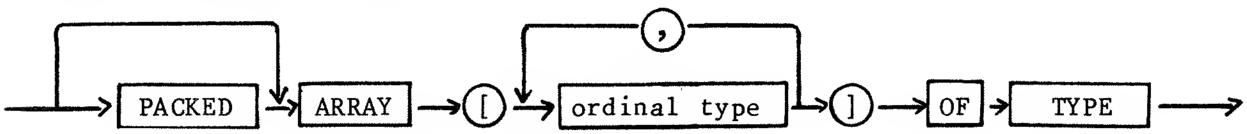
There are four kinds of structured data types: the **ARRAY**, the **SET**, the **FILE**, and the **RECORD**. These four kinds of data types represent four different ways of organizing the simple data types into a data structure. Because a data structure can include as its components other data structures, it is possible to build very complex structures from the basic simple data types.

All structured data types can be packed, by programmer request. The most compact form of storage possible will be used for a packed data structure. Packing a data structure can sometimes save memory. However, packing may cause access time to increase, and so the decision to pack or not depends on the specific application. The keyword **PACKED** signals the compiler to pack the data type into its most compact form. When structured data types contain other structured data types, the keyword **PACKED** must be applied to the innermost structure as well as the outermost to have any effect.

1. THE TYPE ARRAY

An **ARRAY** is a structure composed of a fixed number of data elements which are all of the same type. Data elements within the array can be defined to be of any one type. They can be defined as one of the simple types or as one of the structured types, including **ARRAY.** Arrays can be defined to be of any dimension. (For example, an array of numbers in a row is a one-dimensional array; an array of rows and columns is a two-dimensional array; and so forth.)

Syntax of the type **ARRAY**:



Example declarations:

```
TYPE     table = ARRAY [0..5,1..10] OF INTEGER;
          colors = (red, blue, green, yellow);
VAR       report : ARRAY [1..20] OF table;
          day    : ARRAY [1..365] OF REAL;
          class  : ARRAY [0..8,0..5] OF INTEGER;
          chart  : ARRAY [colors] OF INTEGER;
```

Member elements of the array are specified in the array's index definition (the portion shown in brackets above). The index definition specifies the number of dimensions, the number of elements in each dimension, and how the elements are to be accessed. For example, **ARRAY [0..5,1..10] OF INTEGER** in the example above specifies a two-dimensional array. The first dimension of this array contains six elements (the integers 0 - 5) which are to be accessed in the order 0, 1, 2, 3, 4, 5. The array **chart**, declared as **ARRAY [colors] OF INTEGER**, is a one-dimensional array consisting of the elements **red**, **blue**, **green** and **yellow**, which are to be accessed in the order shown in the type definition.

As shown by the example of the array **chart**, the index definition may consist of a list of ordinal types (excluding the type **INTEGER** for the reason that this would create an array too large to fit in memory). In this case, the number of types specified corresponds to the number of dimensions of the array.

To access one or more elements of a variable declared as type **ARRAY**, first specify the variable name. Then list expressions which evaluate to ordinal values that are within the range of the ordinal types of the index definition.

Examples of accessing array elements:

report[5,3,6] **day[40]** **class[0,0]** **chart[red]**

NOTE: Remember that (. is read by Pascal as equivalent to [and that .) is read as equivalent to]

2. THE TYPE SET

A set is a collection of distinct elements which are all of the same ordinal type. The elements of a set are called set members. There may be up to 256 members in a set. The 256 member limit causes the restriction that a set can not be defined to be of ordinal type **INTEGER.** Also, subranges of type **INTEGER** which include negative integers are not allowed as set base types. A set can have zero members, in which case it is called an empty set.

Syntax of the type **SET**:

```
———→ [SET] ——→ [OF] ——→ [ordinal type] ———→
```

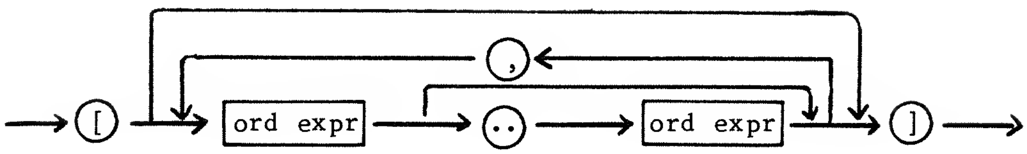
Example declarations:

```
TYPE    days = (sunday, monday, tuesday, wednesday,
                thursday, friday, saturday);
VAR     lowercase, digits, special : SET OF CHARS;
        schooldays, workdays       : SET OF days;
        day                        : days;
```

A variable declared as type **SET** can take on any values which are subsets of the values defined as the type of the set. (This includes the empty set.) For example, the variable **schooldays** declared above might take on the values **monday, tuesday, wednesday, thursday, friday**, a subset of the type **days.**

Set values are denoted by listing set members within square brackets. The individual members can be specified as ordinal expressions.

Syntax of set notation:



The ".." notation between two members specifies that all values in between are also to be included as members. For example, [0..3,7..10] would denote a set with members 0,1,2,3,7,8,9,10. The empty set is denoted by [].

Example assignments to set variables:

```
schooldays:= [monday, wednesday, friday];
workdays   := [monday..friday];
lowercase  := ['a'..'z'];
digits     := ['0'..'9'];
special    := ['*' , '%' , '@'];
```

The relational operators which are applicable to sets are the following, discussed below:

IN	=	< >	< =	> =

IN

A single element can be tested to see if it is a member of a set. The operator **IN** is used for this testing of set membership. This operation evaluates to TRUE if the single element on the left is a member of the set on the right.

=

Two sets can be compared to see if they contain exactly the same members. The operator = is used to test for set equality. If each member of each set is also a member of the other, then the operation evaluates to TRUE.

< >

Two sets can be compared to see if they do not contain exactly the same members. The operator < > is used to test for set inequality. If any member of either set is not also a member of the other, then the operation evaluates to TRUE.

< =

A set can be compared to another set to see if the first set is a subset of the second set. The operator < = is used to test for set inclusion. If all the members of the set on the left are also members of the set on the right, then the operation evaluates to TRUE.

>=

A set can be compared to another set to see if the first set is a superset of the second set. The operator >= is used to test for set containment. If there are no members in the set on the right which are not also members of the set on the left, then the operation evaluates to TRUE.

Example use of these relational operators with sets:

```
IF day IN workdays THEN gotowork; (*gotowork is a procedure*)
IF character IN digit THEN WRITE(character);
If workdays >= schooldays THEN noweekendclasses;
```

Relational Expression	Evaluation
monday IN [monday, tuesday]	TRUE
'A' IN ['a'..'z']	FALSE
'A' IN ['A'..'Z']	TRUE
[1, 2, 3] >= [0]	FALSE
[1, 2, 3] >= [2]	TRUE
['%'] <= ['*', '%']	TRUE
[] <= [tuesday]	TRUE
['a','f','g'] = ['a','f','k']	FALSE

The arithmetic operators which are applicable to sets are described below. When arithmetic operations are performed on two sets, these sets should be of compatible types (that is, they must be of the same base type, or both subranges of the same base type, or one must be a subrange of the other).

+

Two sets can be combined to form a third set containing all the elements that are members of either set. The operator + performs the union of two sets.

-

A set can be formed as the difference between two sets. The operator - performs set difference. The result is a set containing all members of the set on the left which are not also members of the set on the right.

*

A set can be formed which contains only the members which exist in both of two other sets. The operator * performs the intersection of two sets.

Examples:

Expression	Result
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]
[1, 2, 3] + [2, 3, 4]	[1, 2, 3, 4]
[1, 2, 3] - [2]	[1, 3]
[1, 2, 3] - [4]	[1, 2, 3]
[1, 2, 3] * [4, 5, 6]	[]
[1, 2, 3] * [2, 3, 4]	[2, 3]
[red, blue, yellow] + [pink]	[red, blue, yellow, pink]

3. The Type FILE

The data type **FILE** provides the link between a program and the peripheral equipment of the computer system. Variables declared as type **FILE** represent logical files. Input and output operations always refer to logical files. Each logical file has an associated physical file. A physical file is a device such as the computer screen or keyboard, a printer, a disk file, or the "dummy" file. Since all input and output operations reference logical files rather than physical files, a program's input or output can be redirected simply by associating the logical file with a different physical file. (This association is accomplished at the beginning of program execution, when Pascal prompts the user to identify input and output files.)

File data elements can be of any type except **FILE**, and may also not be of structured types containing a component of type **FILE**.

Structured variables (for example, variables of type **ARRAY** or **RECORD**) may contain components of type **FILE**. However, the Input/Output routines (such as **WRITE** and **READ**, discussed in Chapter Ten) will accept only simple variable names. Filenames such as **customer.file1** or **files[2]** are not accepted by the I/O routines. See the Appendix to this guide for an example of how to work around this restriction.
Syntax of type **FILE**:

You can greatly simplify input and output by declaring variables as files of structured types. For example, a complete record can be read or written to a file of records simply by specifying the file variable name and the record variable name as parameters to an input or output procedure.

Example file declarations:

```
TYPE      sales = RECORD
          salesman    : PACKED ARRAY[1..20] OF CHAR;
          quantitysold: INTEGER;
          END;
VAR       salesfile : FILE OF sales;
          numbers   : FILE OF INTEGER;
```

The data elements of files declared as above are read and written in binary (base two) format. Binary format is the form in which data is actually stored in memory. During the I/O process, the data is not translated to a character-readable form. The advantage of this type of I/O is speed of data transfer, and minimized data storage requirements. The obvious disadvantage is that data is in a non-readable form.

Next we'll discuss a special type of file that handles character-formatted data, which you can read. In a **TEXT** file, data is stored as characters. Input and output then involves a translation to and from the binary format used by the computer.

3.a. The Type TEXT

Files of the predefined type **TEXT** have special characteristics. Unlike other file types, a text file is divided into lines, each line being a sequence of characters. The data types which can be input to, and output from, text files, are not restricted to characters only, even though a text file is actually a file of characters. The characters of a text file may represent string, integer, real, or boolean values.

Pascal I/O routines make the appropriate character-to-binary and binary-to-character conversions with files of type **TEXT.** There are two predeclared variables of type **TEXT**, **INPUT** and **OUTPUT.** These variables are used as the default files for input and output procedures and functions, as described in Chapter Ten.

Example declarations:

```
VAR       infile, outfile  : TEXT;
```

There are two procedures and one function built into Pascal which apply only to text files:

WRITELN

The procedure **WRITELN** terminates the current line and positions a file pointer to the next line. If any variables are specified to be output by the **WRITELN**, they are output first and then the file pointer is advanced to the next line.

READLN

The procedure **READLN** causes the file pointer to be positioned to the beginning of the next line. If any variables are specified to be input by the **READLN**, they are input first and then the file pointer is advanced to the next line.

EOLN

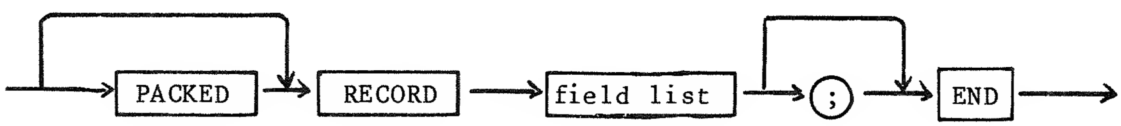
The function **EOLN** is a boolean function which evaluates to TRUE when the end of a line has been reached. At all other times, it evaluates to FALSE.

These I/O procedures and functions are discussed further in Chapter Ten.

4. THE TYPE RECORD

The type **RECORD** is characterized by a fixed number of elements which are called fields. The fields of a record can be of different types. Record field identifiers can be declared to be of any type, including **RECORD.** Therefore, records can be nested.

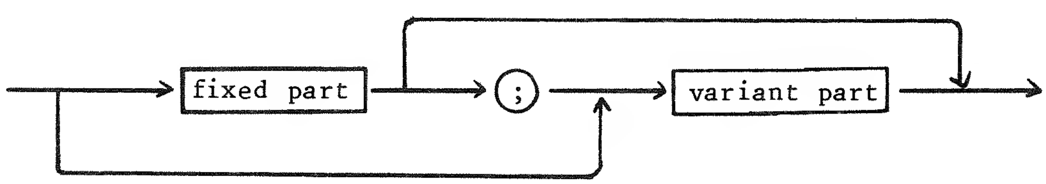
Syntax of the type **RECORD**:



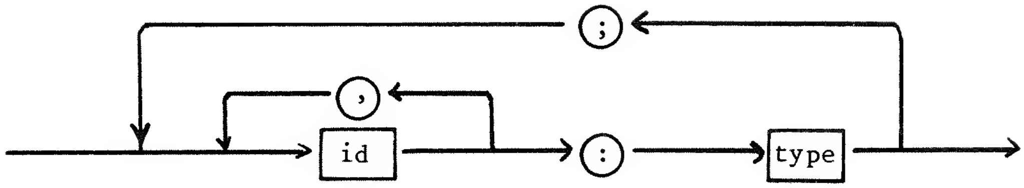
The field list describes the individual components ("fields") of a record. All field identifiers within a record must be unique. However, field identifiers are limited in scope to the record itself, which means that an identifier outside the record definition can be identical to a field name within the record.

The field list consists of two separate parts, a fixed part and a variant part. The fixed part refers to the part of the record which is always referenced in the same way; that is, the fields are fixed. The variant part refers to the part of the record which may be referenced in multiple ways; that is, the fields may vary. Exactly what these two parts are will become clear as the discussion progresses. A record can contain either or both of these parts. If both parts are present, the fixed part must precede the variant part.

Syntax of field list:



Syntax of the fixed part of a record:

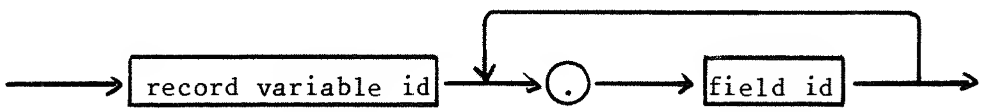


Example record using fixed fields:

```
RECORD
     business : PACKED ARRAY[1..25] OF CHAR;
     location : RECORD
                  street,
                  city,
                  state    : PACKED ARRAY [1..15] OF CHAR;
                  zip      : INTEGER;
                  END;
END;
```

A particular field of a record variable is referenced by the variable name followed by the field name, with a period separating the two names. If the field name is itself a record, then a field within the nested record is referenced by appending a period and the field name to the other two names.

Syntax of record variable referencing:



Example referencing:

Assume that the customer is a record variable as defined above.
Then:

customer.business (references first field)

customer.location (references second field)

The nested fields of the field location are referenced by:

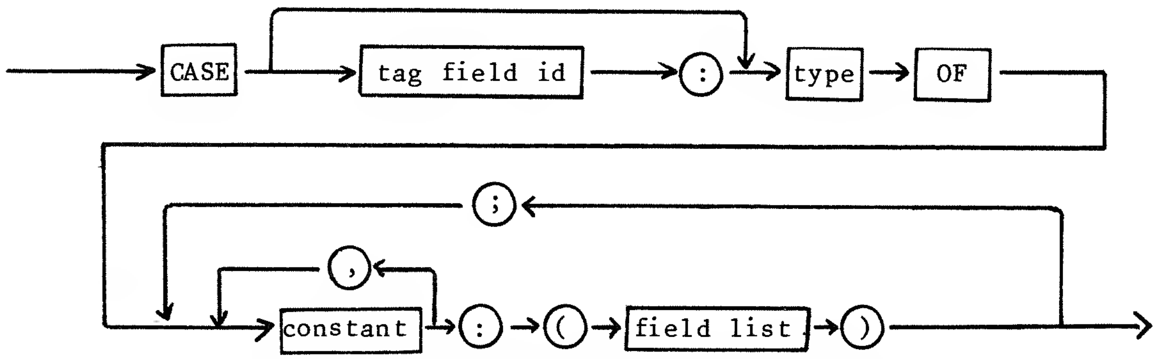
```
customer.location.street
customer.location.city
customer.location.state
customer.location.zip
```

4.a. Record Variants

Sometimes it is useful to be able to define a storage area in a record which can be accessed in multiple ways. Record variants provide the ability to do this. In certain applications, they can simplify a program and save storage space at the same time.

A record variant defines a fixed-size storage area of a record, which can be accessed in multiple ways. The size is determined by the variant alternative (alternative way of accessing the variant) that requires the largest amount of storage space. The variant is defined using a form similar to that of the **CASE** statement.

Syntax of the variant part of a record:



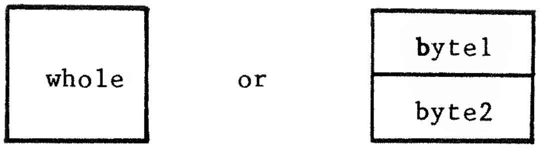
Each alternative way of accessing the storage area of a variant is defined by a field list. All field names within the variant definition must be unique. The storage area can then be accessed in the desired way simply by specifying the appropriate field name.

There are two forms of the variant. In one form, a tag is specified and becomes a field in the record. The tag field resides in the record just prior to the variant storage area. The purpose of the tag field is to store a value which specifies for each record the alternative of the variant that is in effect. The other form omits the tag field, which in some cases is not needed.

Example using no tag field:

```
PACKED RECORD
     CASE BOOLEAN OF
          FALSE: (whole          : INTEGER);
          TRUE : (bytel, byte2   : 0..255;);
END;
```

This variant definition defines a storage area of two bytes (assuming an integer is 16 bits) which is the largest amount of storage required for either of the two field lists. You could then access the whole two-byte storage area as an integer, or you could access each individual byte of the integer. The storage could be represented as follows:



The type **BOOLEAN** was chosen as the selector of the **CASE** because it defines two possible values, which is what is needed to specify the two alternatives. Another type could have been defined and used just as well. With the variant defined as above, you could now reference either the integer or the separate bytes simply by specifying the appropriate field name: **whole**, **byte1**, or **byte2.** For example, if **number** is a variable declared as this record type, then the possible ways of referencing this storage area are:

```
number.whole
number.byte1
number.byte2
```

NOTE: Care should be taken when using variants for this purpose. The way in which the fields of the different forms of the variant overlap one another depends on your implementation (the computer you are using). Also in the above example, which would be the low byte and which would be the high byte is inplementation-dependent.

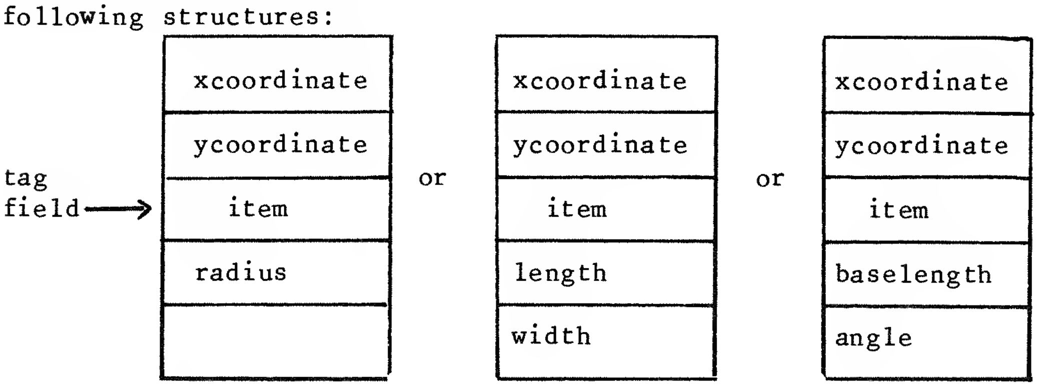
Example Using Tag Field:

Assume the type definition:

```
itemtype = (circle, rectangle, triangle);
```

```
PACKED RECORD
     xcoordinate, ycoordinate : REAL;
     CASE item    : itemtype OF
        circle    : (radius                    :REAL);
        rectangle: (length, width              :REAL);
        triangle : (baselength                 :REAL;
                     angle                     :INTEGER);
     END;
```

This record definition contains a fixed part as well as a variant part with a tag field. The storage allocation for this record could assume the following structures:



The storage allocated would be the amount required to store the two real numbers of the fixed part, the tag field, and the two real numbers of the rectangle field list. The other field lists of the variant require less storage than the rectangle list. Information on which alternative of the variant is in effect can now be stored as part of each record via the tag field. The tag field is referenced in the same way as the other fields.

NOTE: Variants can be nested. That is, a variant can contain a definition of another variant. However, there can be only one variant at any one level, and the variant definition must follow any fixed fields of a record.

CHAPTER FIVE: THE POINTER DATA TYPE

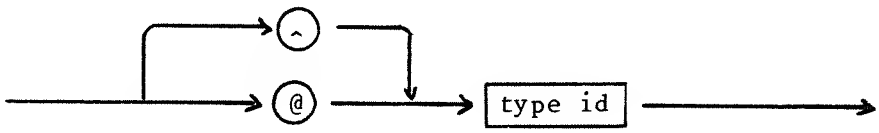
The pointer data type is used in dynamic storage allocation (the creation of storage space for variables during program execution, as that space is needed). Dynamic storage allocation is particularly useful when the amount of data storage a program will require is unknown.

Variables for which storage is dynamically created cannot be referenced in the usual manner. The reason is that they actually have no identifiers of their own. Instead, they are referenced through the use of pointers. A pointer is a variable which points to the location in memory of a dynamically created variable.

When the programmer defines a pointer data type, he or she specifies the data type for which storage will be allocated. The data type specified determines the amount of storage required for each allocation. The definition of a data type does not have to precede the definition of a pointer type which references it. This is the only exception to the rule that identifiers must be defined before they are used. This exception makes it possible for a field of a record to be declared as a pointer to the record itself.

Either the symbol ^ or the symbol @ may be used to signify a pointer type.

Syntax of type pointer:



Example pointer declaration:

```
TYPE transptr = @transaction;
      transaction = RECORD
                     item          :INTEGER;
                     price         :REAL;
                     link          :transptr;
                     END;
```

In the above declaration, **transptr** is a pointer type defined as a pointer to the data type **transaction.** **Transaction** is a record consisting of three components (**item**, **price**, and **link**). Dynamic variables of the type **transaction** can be created through the use of pointer variables of type **transptr.** Notice that **link** is declared to be of type **transptr.** **Link** is a pointer variable which may point to another dynamic variable of type **transaction.** Therefore, a linked list of transaction records can be formed, with the **link** field of each record pointing to the next record.

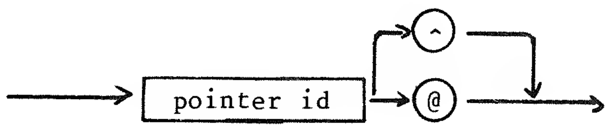
The procedure **NEW**, predeclared by Pascal, is used to allocate storage for dynamic variables. It has one argument, which is a pointer variable. The **NEW** procedure allocates the amount of storage required by the data type associated with the pointer, and then assigns the address of the allocated storage to the pointer. The pointer is then used to reference the allocated storage. For example, consider the declaration:

```
list : transptr;
```

The statement **NEW(list)** would allocate the amount of space required to store the three components of a transaction record at some location in available memory, and assign the location in memory to the variable **list**. (The available memory is called the "heap," and its size is set at run time.)

References to a variable which is pointed to by a pointer are made by following the pointer name with either the symbol ^ or the symbol **@**. For example, **list@** would reference the dynamically created transaction record in the above example.

Syntax of referencing dynamic variables:



Example referencing of dynamic variables:

list@	references whole record
list@.item **list@.price** **list@.link**	reference individual fields
list@.link@	reference record pointed to by link field
list@.link@.item **list@.link@.price** **list@.link@.link**	reference individual fields

When a dynamically created variable is no longer needed, it may be disposed of. This serves to free the space consumed by the variable for other uses. The predeclared procedure **DISPOSE** is provided for this purpose. Like the **NEW** procedure it has one parameter, which is a pointer. The **DISPOSE** procedure frees the memory that is allocated to the variable pointed to by the pointer. Referring to the above example, **DISPOSE(list)** would free the amount of memory which was allocated to the dynamic transaction variable.

A predefined constant **NIL** can be used to assign a value to a pointer. Other than using the procedure **NEW**, assignment to the constant **NIL** is the only way of giving a pointer a defined value. If a pointer's value is **NIL**, then it does not point to a dynamic variable. **NIL** is often used with linked lists to give the pointer of the last element in the list a defined value. It provides a way of detecting when the end of the list has been reached.

Example procedures using pointer variables:

```
PROCEDURE create(VAR translist : transptr);

(* Creates a new transaction
      Adds the transaction to the top of a transaction list
      Returns a pointer to the new transaction via translist
      New transaction becomes top of transaction list *)

VAR
      trans              (*new transaction pointer*)
            :      transptr;
      (* note: translist should be initialized to NIL *)

BEGIN
      NEW(trans);        (*create new transaction*)
      trans.link:=translist;  (*new transaction points to old top
                                    of list*)
      translist:=trans; (*new transaction becomes top of list*)
END;                    (*procedure create*)



PROCEDURE   destroy(translist, trans : transptr);

(* Removes the transaction pointed to by trans from the list
      Recovers the memory used by the transaction *)

VAR
      lead,            (*points to next transaction in list*)
      trail            (*saves location of current transaction
                         while lead is advanced to the next transaction*)
                  : transptr;
BEGIN
      lead:=translist;
      While lead <> trans DO (*search for trans*)
            BEGIN
            trail:=lead;          (*save pointer to current transaction*)
            lead:=lead@.link;     (*advance pointer to next transaction*)
            END;
      IF translist <> trans THEN (*check if trans is at top of list*)
            trail@.link:=lead@.link (*link around transaction*)
      ELSE
            translist:=lead@.link;  (*new top of list*)
      DISPOSE(trans);               (*recover memory*)
END;                                (*destroy*)
```

CHAPTER SIX: OPERATORS

There are four categories of operators: arithmetic, relational, Boolean, and type-transfer.

1. ARITHMETIC OPERATORS

The following table lists all the arithmetic operators, the operations they perform, the type of operands which may be used, and the type of result of the operation., Mixed-mode arithmetic is allowed (for example, an integer value may be added to a real value). Also, automatic truncation occurs when an integer variable is assigned to a real value (that is, the fractional part is discarded -- the real value is "truncated" or "shortened" to become an integer value).

Operator	Operation	Type of Operands	Type of Result
+	addition	integer, real	integer, real
	set union	sets of compatible types	same type as the larger set
-	subtraction	integer, real	integer, real
	set difference	sets of compatible types	same type as the larger set
*	multiplication	integer, real	integer, real
	set intersection	sets of compatible types	same type as the larger set
/	division	integer, real	integer, real
DIV	truncated division	integer	integer
MOD	modulus	integer	integer

NOTE: For sets to be of compatible types, they must be of the same base type, or one base type must be a subrange of the other, or they must both be subranges of the same base type.

2. RELATIONAL OPERATORS

All relational operators perform operations which yield Boolean results. The result is always either TRUE or FALSE. In general, both operands of a relational operator must be expressions of identical type, but the types **REAL**, **INTEGER**, and subranges of **INTEGER** may be mixed.

(Relational operations may be performed on any types except files.)

Operator	Result of Operation
=	true if left operand is equal to right
<>	true if left operand is not equal to right
<	true if left operand is less than right
>	true if left operand is greater than right
<=	true if left operand is less than or equal to right
>=	true if left operand is greater than or equal to right

When character strings are compared, the ordinal numbers of the characters composing both strings (that is, the numbers indicating the positions in the ASCII table of the respective characters) are compared to one another until a pair of characters are different or until the end of the strings is reached. If there are no character pairs which differ, then the strings are equal.

For unequal strings, the first pair of characters which differ determine the relationship. The string whose character's ordinal number is the largest is considered to be greater than the other string.

Operation	Result
'abc' = 'cdf'	FALSE
'abc' < 'abd'	TRUE
'bab' > 'adf'	TRUE

The relational operator **IN** is used exclusively to test for set membership. The left operand may be of any ordinal type, and the right operand may be any set of the same ordinal type.

IN	TRUE if left operand is a member of the right. (For more information on sets and set membership, see Chapter Four.)

3. BOOLEAN OPERATORS

The boolean operators, like the relational operators, yield boolean results. The result is always either TRUE or FALSE. The operands of a boolean operator must be boolean expressions.

Operator	Result of Operation
OR	true if either one or both of the operands is true
AND	true only if both operands are true
NOT	true if operand is false

Operation	Result
FALSE OR FALSE	FALSE
TRUE OR FALSE	TRUE
FALSE OR TRUE	TRUE
TRUE OR TRUE	TRUE
FALSE AND FALSE	FALSE
TRUE AND FALSE	FALSE
FALSE AND TRUE	FALSE
TRUE AND TRUE	TRUE
NOT TRUE	FALSE
NOT FALSE	TRUE

4. OPERATOR PRECEDENCE

Operator precedence is the order in which operations take place within expressions. In general, expressions are evaluated from left to right; however, operations of higher precedence are performed before operations of lower precedence. All operators are ranked by precedence. Parentheses have the highest precedence, and may be used to alter the normal order of evaluation. Nested parentheses are evaluated from the inside out.

Following is a list of the operators, arranged by precedence. Operators listed on the same line have equal precedence.

Highest Precedence -->	( )
	+ - when used as unary operators (that is, as positive and negative sign)
	* / **DIV** **MOD**
	+ -
	= <> < > <= >= **IN**
	NOT
	AND
Lowest -->	**OR**

Operation	Equivalent To	Result
8+3*4	8+(3*4)	20
10-8/4*2	10-((8/4)*2)	6
5 MOD 10-5	(5 MOD 10)-5	0
3<2 OR 6>8 AND TRUE	(3<2) OR ((6>8) AND (TRUE))	FALSE
NOT 7*2<5	NOT ((7*2)<5)	TRUE

5. TYPE TRANSFER

The type-transfer operator is used to temporarily change the type of an existing variable. This is useful when there is a need to reference a variable in a manner which would normally not be allowed by Pascal. For example, you might wish to access the lower and upper bytes of an integer variable. The type-transfer operator allows you to access parts of variables. Also, it provides a mechanism for preventing the compiler from checking <u>type</u>. This may be used in some cases where parameters of differing types must be passed to a procedure.

Syntax of type transfer:

$\longrightarrow \boxed{\text{variable}} \longrightarrow (::) \longrightarrow \boxed{\text{type id}} \longrightarrow$

A type-transferred variable may be used wherever a variable is allowed. Regardless of its original type, the type-transferred variable is then accessed according to the type indicated. The type-transfer operator tells the compiler to treat the variable as if it were of the new type. No data conversion takes place. Type transferred variables must adhere to the same type-matching rules as normal variables.

Example use of type-transfer operator:

```
TYPE        byte = 0..#FF;
            integrec =    PACKED RECORD
                          lower, upper    :byte;
                          END;
            pointer = @integrec;
VAR         number : ARRAY[1..10] OF INTEGER;
            integr : integrec;
            address: pointer;
```

Valid type-transfer operations:

```
    integr.lower := number[1]::byte;
    number[1]::byte := integr.lower;
    READ(integr::INTEGER);
    number[5] := address::INTEGER;
    address::INTEGER := 25 + number[3];
```

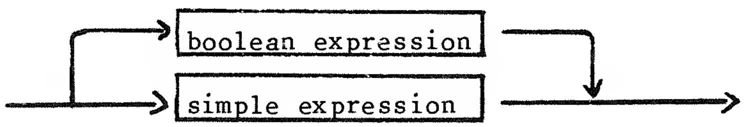
The fundamental use of type transfer is to overlay a type template on a data structure so that components of the structure may be treated as if they were of any desired type. This requires a precise understanding of how the compiler represents the data type (how it is stored) in order to insure that the operation does as it was intended. Because of this, it should be used with caution and only as necessary.

CHAPTER SEVEN: EXPRESSIONS

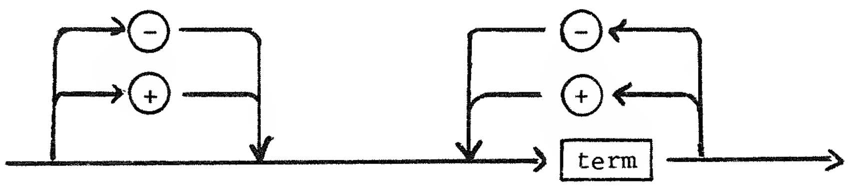
An expression is a variable, a constant, a function call, a set notation, or a combination of these operands with a description of the operations to be performed on them. The operators and operands of an expression define an implicit type for the expression. When evaluated, the expression yields a value of that type.

For example, an integer expression is composed of operands and operators which when evaluated yield an integer result. A real expression yields a value of the type **REAL**, an ordinal expression yields a value which is of one of the ordinal data types, etc. An expression can be just a simple expression (which can yield a value of any data type), or it can be a boolean expression. A boolean expression is composed of simple expressions but always yields a value of the type **BOOLEAN**.

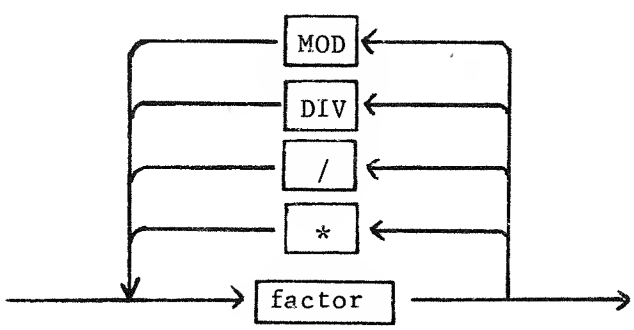
Syntax of expression:



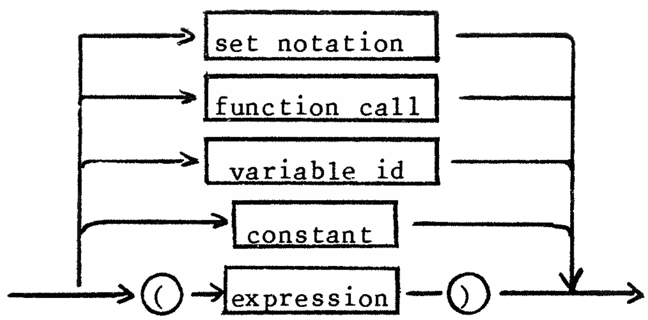
Syntax of simple expression:



Syntax of term:



Syntax of factor:

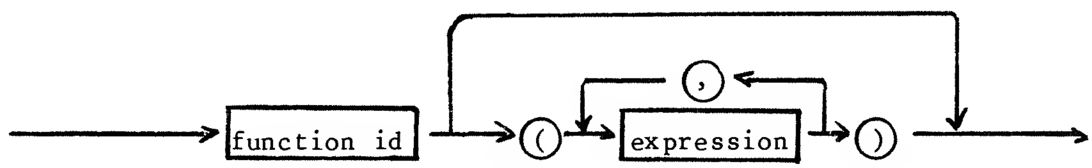


- For the syntax of set notation, refer to the structured data type **SET.**

- A function call has the same form as the **PROCEDURE** statement. The only difference is that a procedure call is a statement while a function call is a part of an expression. Remember that a function has a type associated with it. When a function call is encountered in an expression, the named function is activated. Somewhere in the function, a value is assigned to the function name. When the function terminates, the value assigned to it is substituted in the expression for the function call.

  NOTE: A function may be an ordinal type or the type **REAL** only.

Syntax of a function call:



Example function calls:

salary

payment(interestrate,years)

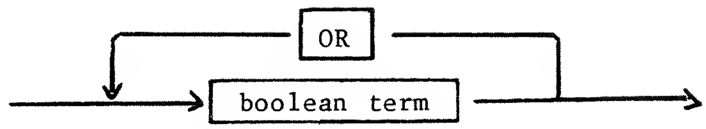
sum(a+b)

Example simple expressions:

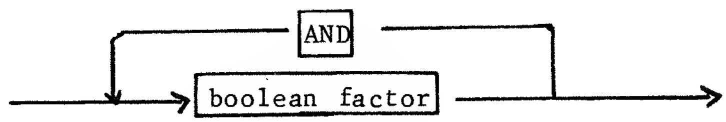
Expression	Result
time	same type as time
weekday + [saturday, sunday]	set
12*payment(interestrate,years)	integer or real depending on the type of function **payment.**

entry MOD size	integer
-10 DIV 4 + 9.2/6-45	real
(var1+var2)*153/(var3-var4)	real

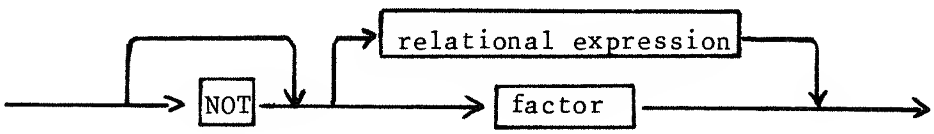
Syntax of boolean expression:



Syntax of boolean term:

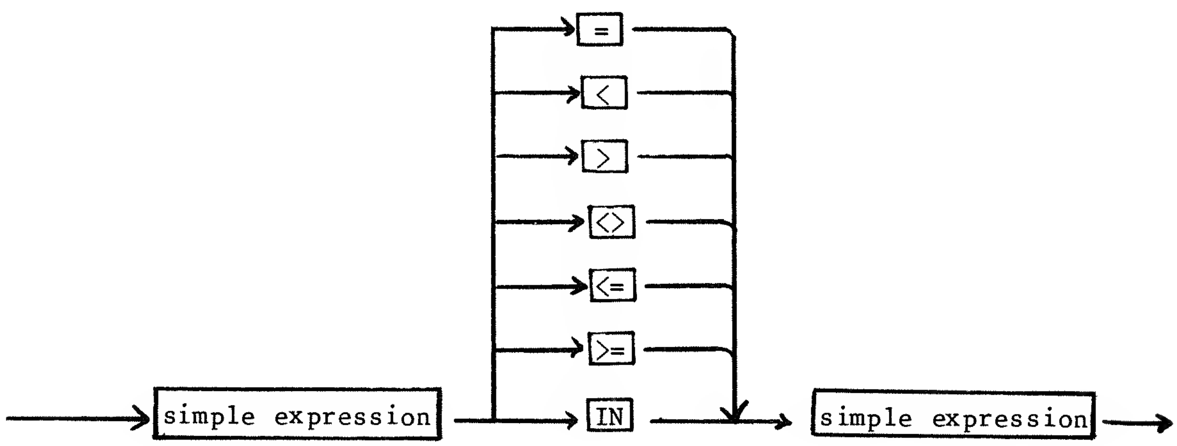


Syntax of boolean factor:



Note: factor must be of type **BOOLEAN.**

Syntax of relational expression:



Example boolean expressions:

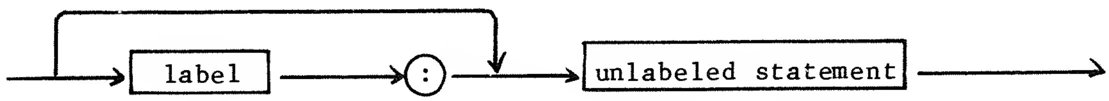
```
a=b OR c<d AND switch

n1 + n2 >= 20 AND n3-n4 <=11

NOT here OR there

NOT alpha < beta AND gamma <> 'R'

number IN [1..15] OR NOT letter IN ['a'..'z']
```

CHAPTER EIGHT: STATEMENTS

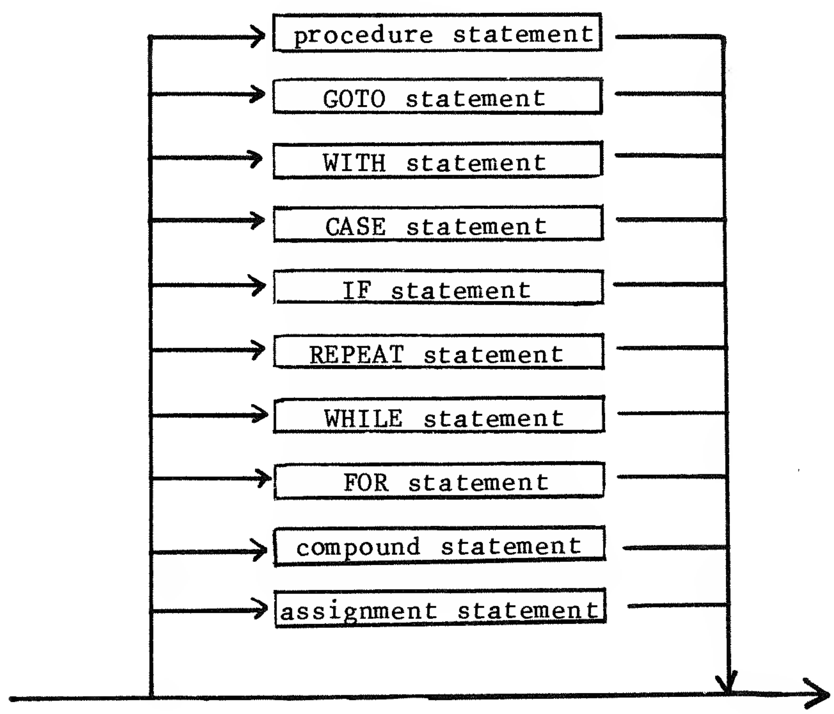
Statements are the Pascal sentences that describe the actions and logic of a program. Statements reside in the statement body part of the block (between and including the **BEGIN** and **END** statements).

A statement may be labeled or unlabeled. A labeled statement may be used in conjunction with the **GOTO** statement. **GOTO** transfers program control to the statement that is identified by the label named in the **GOTO**'s argument. Before a statement can be labeled, the label must be declared in the **LABEL** declaration part of the block in which the statement appears.

Syntax of a statement:



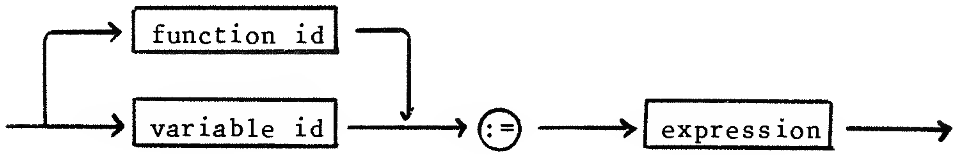
Syntax of an unlabeled statement:



1. THE ASSIGNMENT STATEMENT

The assignment statement is used to assign values to variables and function identifiers.

Syntax of the assignment statement:



The action of the assignment statement is to give the variable or function identifier on the left side of the equal sign, the value of the evaluated expression on the right side. The variable may be of any type. In general, the type of the variable or function must be the same as the type of the evaluated expression. However, there are some exceptions. An identifier of type **REAL** may be assigned a value which is an integer or a subrange thereof. One side may be a subrange of the other, but the value to be assigned should be in the range of the left side. If the identifier on the left side is a **SET** type, it may be assigned to a set which differs in type as long as the set members of the right side are allowable members of the set on the left side.

Example assignment statements:

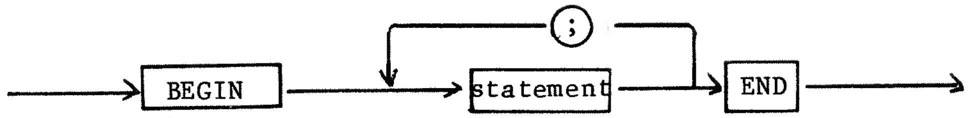
Assignment	Left-hand side identifier type
a := 10	integer or real
x := 100.5 + 49 + 87/12	real
y := abs(10*z-30.3)	real
test := sample < 10	boolean

2. THE COMPOUND STATEMENT

Statements which are bracketed by the two keywords **BEGIN** and **END** make up what is termed a compound statement. The compound statement is used in places where more than one statement is required. The compound statement is essential for most of the control structures of Pascal. For example, the **FOR** statement is a control structure used for executing a statement repeatedly for a specified number of times.

The compound statement provides the ability to use this construct for executing a sequence of statements rather than just one.

Syntax of the compound statement:



Example compound statement:

```
BEGIN
   a := b * c;
   d := a/10 + 16.9;
   e := d - 28.3 + 14;
END
```

3. REPETITIVE STATEMENTS

Repetitive statements are structures used for loop control. They specify that a statement or sequence of statements is to be executed repeatedly until some terminating condition occurs. Pascal provides three such control structures.

3.a. The FOR Statement

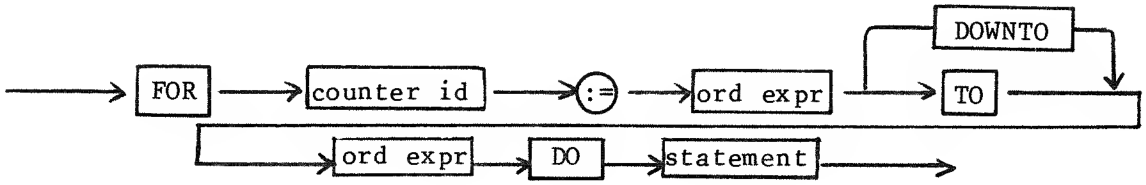
The **FOR** loop is used when a statement is to be executed a predefined number of times. The **FOR** loop is characterized by a loop variable which serves as a counter for controlling the number of times a statement is executed.

The counter has defined starting and ending values which are ordinal expressions. The expressions are evaluated once upon entry into the loop. At the beginning of each time through the loop, the counter's value is compared to the ending value to determine whether or not to end execution of the **FOR.** At the end of each time through the loop, the counter's value changes by 1. If the keyword **TO** is used, the counter is incremented each time through the loop. Use of the keyword **DOWNTO** causes the counter to be decremented.

The loop is terminated when the counter has incremented or decremented past the ending value. The **FOR** statement is not executed if the counter's starting value is such that the ending value would never be reached. For example, if the starting value was -1, the ending value was 2, and **DOWNTO** was used, the **FOR** statement would not be executed.

NOTE: The compiler option **FORDECL** may be used to cause the compiler to generate temporary variables for **FOR** loop counters. When this option is used, it is not necessary to declare the counter variable.

Syntax of the **FOR** statement (counter must be of an ordinal type):



Example **FOR** statements:

```
FOR i := 1 TO 30 DO WRITELN(' This gets written 30 times');

FOR j := first DOWNTO last DO
      BEGIN
      initialscore[j] := 0;
      time[J] := 60;
      END
```

3.b. The WHILE Statement

The **WHILE** statement uses a boolean expression to control repeated execution of a statement.

Syntax of the **WHILE** statement:

The evaluation of the boolean expression precedes the execution of the statement. If the expression evaluates to TRUE, the statement is executed and then the expression is reevaluated. This loop continues until the expression evaluates to FALSE. The first occurrence of a FALSE evaluation causes termination of the **WHILE** statement.

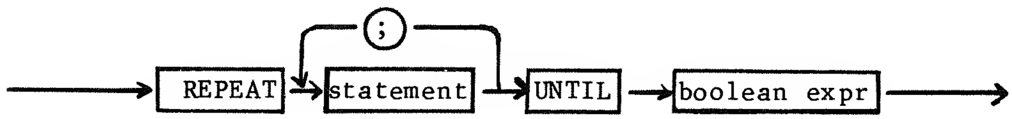
Example **WHILE** statements:

```
WHILE NOT EOLN DO READ(character)


WHILE (a<b) AND (b<c) DO
      BEGIN
      WRITELN(a,b,c);
      a := a + 1;
      c := c - 1;
      END
```

3.c. The REPEAT Statement

The **REPEAT** statement, like the **WHILE** statement, uses a boolean expression to control repeated execution. The major difference is that the **REPEAT** is designed such that a sequence of statements which are bracketed by the two keywords **REPEAT** and **UNTIL** will be executed at least once. Following the keyword **UNTIL** is a boolean expression. If the expression evaluates to FALSE, then execution returns to the first statement following the **REPEAT** keyword. If the expression evaluates to TRUE, then execution continues with the statement following the boolean expression.

Syntax of the **REPEAT** statement:



Example **REPEAT** statement:

```
REPEAT
      i    := i+1;
      j    := j+1;
      k[j] := (i + j) MOD 100;
      l[i] := (i + j) MOD 200;
UNTIL  i=j
```

4. CONDITIONAL STATEMENTS

Conditional statements are used when the execution of a statement must be controlled by some predetermined condition, or when one statement out of a group of statements is to be selected for execution. There are two conditional statements: the **IF** statement and the **CASE** statement.

4.a. The IF Statement

The **IF** statement uses a boolean expression to control the execution of statements.

Syntax of the **IF** statement:

In its simplest form, the **IF** statement involves the evaluation of a boolean expression to determine whether or not to execute an associated statement which follows the keyword **THEN**. If the expression is TRUE, then the statement is executed, otherwise it is not. The **IF** statement can also contain an **ELSE** clause. In this form, if the boolean expression is TRUE, then the statement following the keyword **THEN** is executed; otherwise the statement following the keyword **ELSE** is executed.

Example **IF** statements:

```
IF finished THEN WRITELN(' operation complete');

IF number < 10 THEN range := 1 ELSE range := 2;

IF alpha >= '0' AND alpha <= '9' THEN digit(alpha)
ELSE
     IF alpha >='A' AND alpha <= 'Z' THEN letter(alpha)
     ELSE
        special(alpha);

IF contextlist = NIL THEN
     BEGIN
     NEW(context);
     context@.link := NIL;
     contextlist := context;
     END
ELSE
     BEGIN
     temp := context;
     NEW(context);
     temp@.link := context;
     context@.link := NIL;
     END;
```

The statements following the keywords **THEN** or **ELSE** can themselves be **IF** statements. In some forms, an ambiguity can exist in determining which **ELSE** clause goes with which **IF**. For example, consider the following case in which **b1** and **b2** represent boolean expressions and **s1** and **s2** represent statements.

```
IF b1 THEN IF b2 THEN s1 ELSE s2
```

The **ELSE** could go with the first **IF** or the second **IF**. The rule used for solving the ambiguity is to associate an **ELSE** clause with the nearest **IF**. The above statement would then be equivalent to:

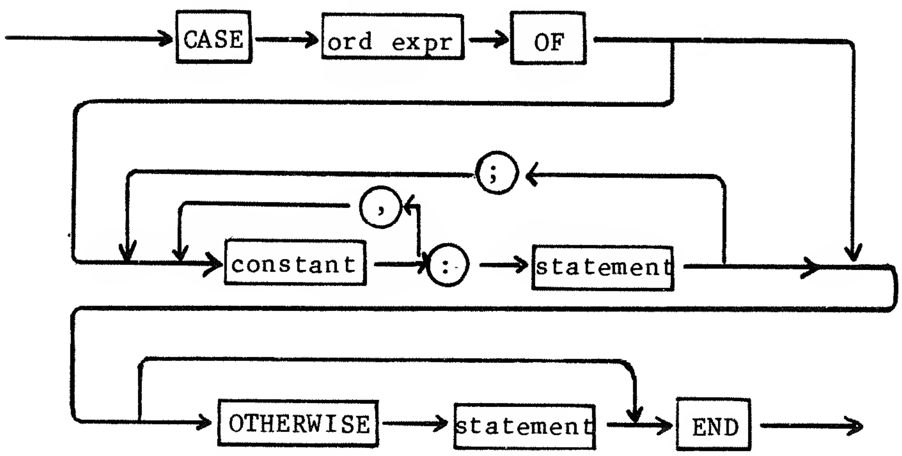
```
IF b1 THEN
     BEGIN
     IF b2 THEN s1 ELSE s2
     END
```

CAUTION: Semicolons must never appear in the middle of a statement, but a common error for beginning programmers is to put a semicolon in an **IF** statement which has an **ELSE** clause. While semicolons are necessary for separation of the individual statements within a compound statement, they must not separate an **ELSE** from its corresponding **IF**.

4.b. The CASE Statement

The **CASE** statement uses an ordinal expression to select one statement out of a group of statements for execution. The group of statements represent alternatives. When a **CASE** statement is executed, one of the alternatives is selected and executed and then control passes to the statement following the **CASE** statement.

Syntax of the CASE statement:



The alternative statements of a **CASE** statement are preceded by constants. The ordinal expression is evaluated and compared to the constants preceding the alternative statements. If a match is found, the statement which has the preceding constant that matches the evaluated expression is executed.

There are two actions that can take place in the event that no match is found. By using the **OTHERWISE** clause, you may specify a statement to be executed when no match is found. If the **OTHERWISE** clause is omitted and no match is found, then execution continues with the statement which follows the **CASE** statement.

Example **CASE** statements:

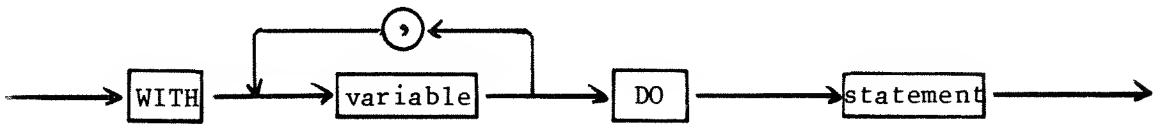
```
CASE n1+n2 OF
   10: x := sin(x);
   11: x := cos(x);
   12: x := ln(x);
END;

CASE ch OF
   'a','b','c': token :=0;
   'd','e','f': token :=1;
   OTHERWISE    token :=2;
END;
```

```
CASE day OF
     monday    : snack  := apple;
     tuesday   : snack  := orange;
     wednesday : snack  := grapes;
     thursday  : snack  := pear;
     friday    : snack  := candy;
     saturday,
     sunday    : BEGIN
                 weekend := TRUE;
                 snack   := nothing;
                 END;
   END;
```

5. The WITH Statement

The **WITH** statement is used in conjunction with variables of type **RECORD.** It makes it possible to use a shorter notation when referencing fields of record variables.

Syntax of the **WITH** statement:



The variable list specifies the record variables whose fields are to be referenced simply by specifying the field name itself. When fields of a record are nested (that is, when a record is defined as a field of another record), the record variable and the fields, down to the level of the field which is to be referenced in short notation, may be specified in the variable list. Then you can reference the nested field in the statement simply by specifying its field name.

There is a conflict inside the **WITH** statement when an identifier corresponds to both a variable name and a field name of one of the specified records. For example, you could have a record variable named **weekday** with a field named **monday**, and also have a simple variable named **monday**. Then the following **WITH** statement might be used:

```
WITH weekday DO monday :=1
```

In such a case, the field name takes precedence over the variable name, and the field of the record is referenced. If nested **WITH** statements are used and a field name inside occurs in more than one of the specified records, then the closest **WITH** takes precedence.

Example **WITH** statements:

Assume these declarations:

```
customer :  RECORD
            name,
            address,
            city      : PACKED ARRAY[1..17] OF CHAR;
            date      : RECORD
                        month,
                        day,
                        year         : INTEGER;
                        END;
            END;
```

```
WITH customer DO
     BEGIN
     name    :=  'JACK SLATE       ';
     address :=  '1216 MELODY LANE ';
     city    :=  'TULSA, OKLAHOMA  ';
     END;

WITH customer.date DO
     BEGIN
     month   :=  10;
     day     :=  23;
     year    :=  1981;
     END;
```

6. THE GOTO STATEMENT

The **GOTO** statement is used to cause an unconditional branch to a labeled statement.

Syntax of the **GOTO** statement:

The label must be declared in the **LABEL** declaration part of the same block which contains the **GOTO** that references it. The **GOTO** statement cannot specify a branch to a label outside the block in which it resides. Care must be taken when using the **GOTO** statement. For example, you should not branch to the inside of a **FOR** loop from a statement outside the loop. This could cause some very unpredictable results.

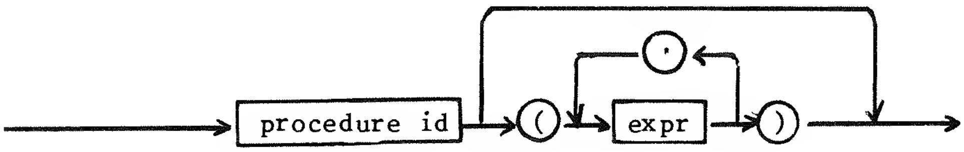
Example **GOTO** statement:

```
FOR i := 1 TO 1000 DO
      IF a[i] <> b[i] THEN GOTO 10
      ELSE a[i] := b[i];
10:  a[i] := '#0D';
```

7. THE PROCEDURE STATEMENT

The **PROCEDURE** statement causes the activation of a procedure. Control passes to the named procedure, and then returns to the statement following the **PROCEDURE** statement when the activated procedure terminates. If a procedure has a parameter list, a procedure statement which activates it must specify an argument for each parameter of the parameter list. The arguments must match the order and type of the parameters specified in the parameter list of the procedure. An argument is specified by an expression. If a parameter of the procedure is a pass-by-reference parameter (denoted by **VAR**), the corresponding argument of a procedure statement must be a single variable name. The variable may be a simple variable or a component of a structured variable.

Syntax of a **PROCEDURE** statement:



Example **PROCEDURE** statement (call):

(See the declaration of procedure **getvalue** in Chapter Two.)

```
getvalue(n+j,8,hexstring,value)

report

writeout(x,y,3.7+9.6/z)
```

CHAPTER NINE: PROCEDURES AND FUNCTIONS

> NOTE: See Chapter Two for a description of the syntax of procedure and function declarations. A discussion of parameter passing is included with the discussion of the procedure heading.

Procedures and functions are the tools used to modularize a program. To modularize a program is to break the program up into small, manageable pieces. Procedures and functions increase the readability of a program and make any later modifications easier to handle.

Procedures and functions can be compiled separately from the main program, and then can be linked to the program(s) that use(s) them. This allows the programmer to develop libraries of commonly used procedures and functions, for use with many programs. Then all the programs that use a given function can link to it rather than having to have the function included in the body of the program.

The variables declared in a procedure or function do not occupy storage space until the procedure or function is activated. When activated, storage space is allocated for variables, and when the procedure or function is terminated the allocated space is released. Therefore, the amount of storage (or stack) space required by a program at any point in time depends on the number of blocks which are activated at that time.

A procedure is activated (or called) by a procedure statement. When a procedure is called, control is passed from the point of the call to the procedure. The statements in the procedure are then executed. When the block **END** of the procedure is reached or when a call to the **ESCAPE** procedure is made, control passes back to the statement following that which activated the procedure.

A function is activated by an expression. When an expression which contains a reference to the function is evaluated, the function reference causes control to pass to the named function. The statements in the function are then executed. Unlike procedures, functions have a declared type. At some point inside the statement body of a function, the function name should be assigned a value. The value must be the same type as the type to which the function is declared. When the block **END** of the function is reached or when a call to the **ESCAPE** procedure is made, control passes back to the evaluation of the expression which activated the function and the function reference is replaced by the value assigned to the function.

1. SCOPE RULES

A procedure or function declaration forms a new block which is a subblock of the block in which the declaration appears. The new block formed is nested within the block which declares it. This process of nesting which occurs every time a procedure or function is declared produces a program structure such as the one shown on the first page of Chapter Two.

Any block which is enclosed by another block is said to be nested within that block. The level numbers on the diagram indicate how deep the nesting goes beyond the program block which is arbitrarily assigned level one. The existence of procedures and functions makes it necessary to talk about scope rules. Scope rules describe the accessibility of identifiers from any particular place in a program. The two terms local and global are helpful in discussing scope rules.

An identifier is considered to be local to a block if the identifier is declared within the same block. If there are no blocks nested within the declaring block, then a local identifier can only be referenced by the block which declares it. Enclosing blocks cannot access a local identifier.

An identifier is considered to be global to blocks which are nested within the block in which the identifier is declared. If an identifier is global to a particular block, then that block can reference the identifier, provided that it has not declared an identifier of the same name. If a block declares an identifier with the same name as a global identifier, then the global identifier is no longer accessible from that block. Also, any further nested blocks will not have access to the original global identifier.

Identifiers declared in the program block are accessible from any place in a program because all other blocks are nested within the program block. Therefore, identifiers declared in the program block are global to all procedures and functions of the program. Identifiers declared in a procedure or function are local to that procedure or function. The only place in the program which can access these identifiers are the procedure or function itself and the procedures or functions, if any, which are nested within. The nested procedures or functions can access only the global identifiers which they do not declare themselves.

A procedure or function declaration consists of a heading followed by a block. It is important to note that the procedure or function name of a heading is local to the block which declares it. The parameters of the heading are local to the procedure or function itself. This means for example that a procedure statement in the program block can reference any procedure declared in the program block. However, a procedure statement in the program block cannot reference any procedure declared within one of these procedures.

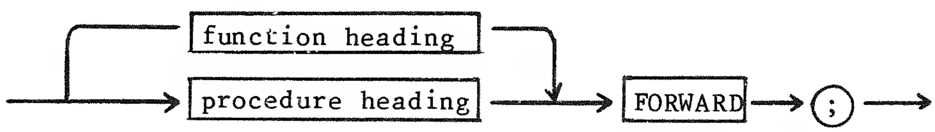
As an example of how scoping affects the accessibility of identifiers, consider the sample diagram on the first page of Chapter Two. The following table shows for each block of the diagram, the procedures and functions which may be called from that block, and the constants, types, variables, etc., which can be referenced by the block.

Block	accessible procedures and functions	accessible constants, types, variables, etc.
A	B, D, F	A
B	B, C, D, F	A, B
C	B, C, D, F	A, B, C
D	B, D, E, F	A, D
E	B, D, E, F	A, D, E
F	B, D, F	A, F

2. FORWARD

The rule that an identifier must be declared before it is referenced means that a procedure or function must be declared before it is referenced by a procedure statement or by an expression with a function reference. Some calling sequences that occur among a group of procedures or functions make it impossible to obey this rule. For example, if two procedures call each other, then you cannot declare one without referencing the other. The keyword **FORWARD** provides the mechanism for getting around this problem. Using the keyword **FORWARD** with just the heading for a procedure or function declaration signals the compiler that the procedure or function block will be declared at some later point in the program. If the procedure or function has parameters, the parameters are declared as well. Then the procedure or function which has been forward declared may be referenced.

Syntax of forward declaring a procedure or function:



The actual declaration of a forward declared procedure or function can appear at some later place in the program. The place that it appears must be at the same level and scope as its forward declaration. The actual declaration consists of the heading with no parameters, followed by the block. Since the parameters were declared in the forward declaration, they must not be declared again in the actual declaration.

If a forward declared procedure or function does not have its actual declaration present, then it is treated as an external procedure or function.

Example use of **FORWARD**:

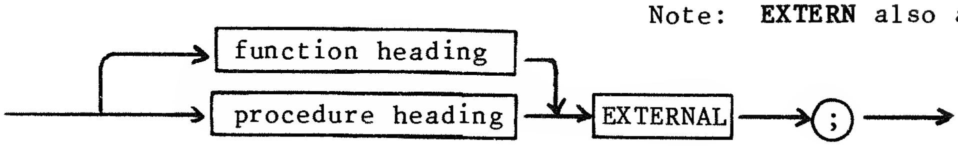
```
PROCEDURE abc(pl, p2 : INTEGER); FORWARD;

PROCEDURE xyz;
VAR    pl, p2 : INTEGER;
BEGIN
abc(pl,p2);
END;

PROCEDURE abc;
BEGIN
. . . . .

END;
```

3. EXTERNAL

An external procedure or function can be declared in a program by specifying its heading followed by the keyword **EXTERNAL**.

Syntax of externally declared procedures or functions:

Note: **EXTERN** also accepted.



NOTE: For brevity, the word "routine" will be used in place of "procedure or function" in the following discussion.

The linking loader may be used to link separately compiled routines to a program. When a routine is declared to be external, its actual declaration does not have to appear in the program. This is very useful when you are working with large programs. A large program may be broken up into many routines which are declared as external. The external routines can then be compiled individually. The linking loader can then be used to link the compiled program to its individually compiled routines. One advantage to this is that any changes which are made to a particular routine will cause only that routine to have to be recompiled. The linking process is then repeated after the changed routine has been recompiled. Another advantage is that slightly larger programs can be created by compiling them in pieces and then linking the pieces together.

Perhaps one of the most frequent uses of external routines is to create a file or library of commonly used routines. The all the programs which use the routines can link to them rather than having to declare them in each program.

A compiler option must be used to compile a routine by itself. The reason is that a routine by itself is not a legal Pascal program. Therefore, a legal program must be constructed around the routine. This would include a program heading, the environment of the routine, the procedure or function declaration, and a statement body. The environment consists of any constants or types which are in the scope of and are used by the external routine. If global constants or types are needed by the routine, they should be given the same names as those used in the programs that use the routine. The scope refers to the identifiers in a program which are accessible to the externally declared routine.

Variables can also be included in the environment but this is not recommended. If an external routine needs to access a global variable, the variable should be passed as a parameter to the routine. Otherwise, extreme care must be taken to assure that the environment around the external routine matches the environment of the programs which use the routine. The statement body contains the compiler option which is called **NULLBODY**. The **NULLBODY** option tells the compiler not to generate any code for the program. Only code for the declared routine is generated.

The syntax for using the **NULLBODY** compiler option is shown in the appendix to this guide, along with all the other compiler options. An example using global variables in an external procedure is also given.

Example use of external procedure:

```
PROGRAM sample;
CONST . . . . .
TYPE  . . . . .
VAR   xmin,xmax,ymin,ymax : REAL;
      . . . . .

PROCEDURE axes(xmin,xmax,ymin,ymax : REAL); EXTERNAL;
BEGIN
      . . . . .
  axes(xmin,xmax,ymin,ymax);
      . . . . .
END.      (*sample*)
```

Separate compile of procedure **axes**:

```
PROGRAM axesroutine;
(*global environment, if any, goes here*)
PROCEDURE axes(xmin,xmax,ymin,ymax : REAL);
  TYPE  . . . .
  VAR   . . . .
  BEGIN
      . . . .
      . . . .
  END; (*procedure axes*)
BEGIN
  (*$NULLBODY*)
END.
```

4. RECURSION

Pascal is a language which supports recursion. Recursion refers to having more than one activation of a particular procedure or function at the same time. There are two forms of recursion. Direct recursion refers to a procedure or function that calls itself. Indirect recursion refers to a procedure or function that makes a call which eventually results in the procedure or function being called again. An example of this is two procedures which call each other. When writing recursive procedures, some conditional statement must exist in the procedure to halt the recursion at some point. Otherwise, there would be an endless loop that would terminate only after the stack was exhausted, crashing the program. Recall that each activation of the procedure results in space being allocated for its variables.

Example use of recursion:

```
PROCEDURE XYZ;
   (*DECLARATION HERE*)
BEGIN
. . . . .
   XYZ;  (*PROCEDURE CALLS ITSELF*)
. . . . .
END;
```

5. PREDECLARED PROCEDURES AND FUNCTIONS

The predeclared procedures and functions are accessible from any place in a program. They are declared in an imaginary block which surrounds the program block. The names of predeclared procedures or functions may be used as identifiers in programs. This means that the name of a predeclared procedure or function may be used in a declaration. If so, then the predeclared procedure or function whose name is used in a declaration is no longer accessible to the program. Its name is associated with the new declaration.

File-Associated Procedures (Replace f with a file specification.)

RESET(f)	Positions the file pointer of the specified file to the beginning for the purpose of reading. If the file is empty, then the function **EOF** becomes true, otherwise it is false.
REWRITE(f)	Replaces the specified file with an empty file. The file pointer is positioned to the beginning of the file.
PAGE(f)	Outputs a formfeed to the specified file. Formfeeds cause skipping to the top of the next page when the file is printed.

CLOSE(f)	Closes the specified file. This procedure may be used to explicitly close a file at any time.
MESSAGE(s)	Outputs the specified string to the terminal. **S** is **CHAR** or **ARRAY OF CHAR.**
READ, READLN	Read data from a device.
WRITE, WRITELN	Write data to a device--see Chapter Ten for details.

Arithmetic Functions

	Operation	Type of x	Type of Result
ABS(x)	absolute value	**INTEGER, REAL**	same type as **x**
SQR(x)	square	**INTEGER, REAL**	same type as **x**
SIN(x)	sine	**INTEGER, REAL**	**REAL**
COS(x)	cosine	**INTEGER, REAL**	**REAL**
ARCTAN(x)	arctangent	**INTEGER, REAL**	**REAL**
EXP(x)	natural (base e) exponential	**INTEGER, REAL**	**REAL**
LN(x)	natural logarithm	**INTEGER, REAL**	**REAL**
SQRT(x)	square root	**INTEGER, REAL**	**REAL**

Boolean Functions

ODD(x)

Operation: Returns true if **x** is odd, else false.
Type of **x**: **INTEGER**
Type of result: **BOOLEAN**

EOLN(x)

Operation: Returns true if the end of a line in in the file has been reached.
Type of **x**: **TEXT**
Type of result: **BOOLEAN**

EOF(x)

Operation: Returns true if the end of the file has been reached.
Type of **x**: **FILE**
Type of result: **BOOLEAN**

Transfer Functions

TRUNC(x)	Operation: Truncates a real value to its integer part. Type of **x**: **REAL** Type of result: **INTEGER**
ROUND(x)	Operation: Rounds a real value to the nearest integer. Type of **x**: **REAL** Type of result: **INTEGER**
ORD(x)	Operation: Returns the ordinal number of **x**. Type of **x**: any ordinal type Type of result: **INTEGER**
CHR(x)	Operation: Returns the character whose ordinal number is **x**. Type of **x**: **INTEGER** Type of result: **CHAR**
LOCATION(x)	Operation: Returns the address of variable **x**. Type of **x**: any type (also may be a procedure name) Type of result: **INTEGER**
SIZE(x)	Operation: Returns the size of type **x** in bytes. Type of **x**: any type identifier Type of result: **INTEGER**
HB(x)	Operation: Returns the high byte of **x**. Type of **x**: **INTEGER** Type of result: **INTEGER**
LB(x)	Operation: Returns the low byte of **x**. Type of **x**: **INTEGER** Type of result: **INTEGER.**

Data Transfer Procedures

PACK(a,i,z)	Operation: Copy the unpacked array **a** into the packed array **z**. If the dimension of **a** is **m..n** and the dimension of **z** is **u..v** and **n-m** > **v-u**, then the operation is equivalent to: for **j:= u** to **v** do **z[j] := a[j-u+1]**
UNPACK(z,a,i)	Operation: Unpacks the above array.

Dynamic Allocation Procedures

NEW(p) — Allocates a new variable **v** and assigns the pointer reference of **v** to the pointer variable **p**. Tag field values may appear as parameters to **NEW**, but are non-functional.

DISPOSE(p) — Releases the storage occupied by the variable pointed to by **p**.

Other Functions

SUCC(x) — Operation: Returns the successor of **x** which is the next higher value in the enumeration of which **x** is a member.
Type of **x**: any ordinal type
Type of result: same type as **x**

PRED(x) — Operation: Returns the predecessor of **x** which is the next lower value in the enumeration of which **x** is a member.

Other Procedures

ESCAPE — Causes termination of a block just as if the block end had been reached. If the block is a procedure or a function, then control returns to the calling block. If the block is the program block, then program execution is terminated.

NOTE: If files are declared logically within a procedure, then the files must be closed using the procedure **CLOSE** before **ESCAPE** is called. Normal termination of a block results in files automatically being closed.

CHAPTER TEN: INPUT AND OUTPUT

Input and output refer to the communication of a program with the external environment. A program communicates with the external environment through the use of logical files. Logical files are the variables in a program which are declared as type **FILE** or **TEXT.** The logical files are then associated with physical files. Physical files are the actual devices of a computer system. A physical file could be a disk file, a terminal, a printer, or some other device. The method of associating logical files with physical files is discussed on pages E-4 and E-5.

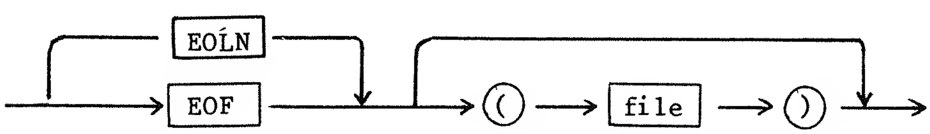
Predeclared procedures and functions are provided for handling input and output. These procedures and functions have a characteristic unlike other procedures and functions. The number of parameters passed to them can vary. They may be called with no parameters or with several parameters. Since each input and output routine performs an operation on a file, it must know which file to operate on. If a routine is passed to the logical file name, then it operates on the specified file, otherwise it operates on a default logical file. The two predeclared variables **INPUT** and **OUTPUT** are the default logical files. They are both declared as type **TEXT.** The one used as the default depends on the routine called. The input routines default to **INPUT** and the output routines default to **OUTPUT.**

I/O Routines

Procedures			Functions
input	output	general	
RESET	**REWRITE**	**CLOSE**	**EOF**
READ	**WRITE**		**EOLN**
READLN	**WRITELN**		
	PAGE		
	MESSAGE		

A file has associated with it a file pointer. The file pointer is used to point to an individual component of a file. There are two predeclared boolean functions which may be used to check the status of a file's pointer. Both functions may or may not take a logical file name as a parameter. If no file parameter is passed, the default is **INPUT.** The function **EOF(file)** returns the value TRUE if the pointer is at the end of the file. Otherwise, the value returned is FALSE. The function **EOLN(file)** can only be used with files of type **TEXT.** It returns the value TRUE when the file's pointer is at the end of a line. Otherwise, the value returned is FALSE.

Syntax of **EOF** or **EOLN:** (default: file = **INPUT**)



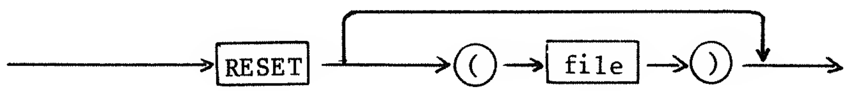
Examples of using **EOF** and **EOLN**:

```
WHILE NOT EOF(datain) DO
   BEGIN
   WHILE NOT EOLN(datain) DO
      BEGIN
      READ(datain,ch);
      . . . . .
      END;
    READLN(datain);
   . . . . .
   END;
```

1. RESET

The **RESET** procedure opens a file so that it can be read. No input can be received from a file (except for the default file **INPUT**) without this operation first being performed.

Syntax of **RESET**: (default: file = **INPUT**)



The **RESET** procedure positions the file pointer to the beginning of the file. If the file is empty, then the function **EOF(file)** becomes TRUE. If the file is not empty, then the function **EOF(file)** becomes FALSE.

The statement **RESET(INPUT)** is implicitly executed at the beginning of a program unless the **NO INOUT** compiler option is used. Therefore, it is not necessary for a program to explicitly open the default logical file **INPUT**.

Example use of **RESET**:

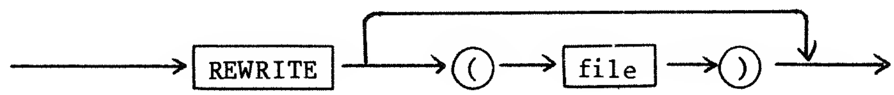
```
PROGRAM readdata;
VAR     datain : TEST;
BEGIN
   RESET(datain);   (*open file datain for reading*)
   . . . . .
END.
```

Input and output to files is buffered. This is to prevent having to access a physical device every time an operation is performed. Each file used by a program has an associated file buffer. In this implementation, the input buffer of a file is not filled until a **READ**, **READLN**, **EOLN**, or **EOF** is performed on the file. This prevents the normal problems associated with reading from a terminal. Programs can have their logical files remapped from a disk file to a terminal without modification to the program itself.

2. REWRITE

The **REWRITE** procedure opens a file so that it can be written. No output can be sent to a file (except for the default file **OUTPUT**) without this operation first being performed.

Syntax of **REWRITE**: (default: file = **OUTPUT**)



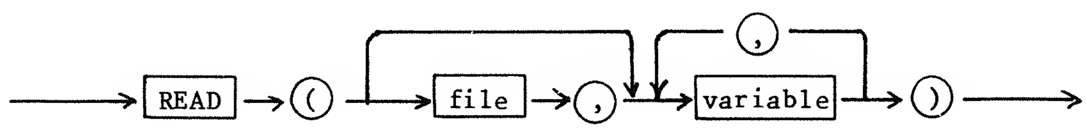
The procedure positions the file pointer to the beginning of the file. The file becomes empty when this happens. This means that any data in the file is lost.

The statement **REWRITE(OUTPUT)** is implicitly executed at the beginning of a program unless the **NO INOUT** compiler option is used. Therefore, it is not necessary for a program to explicitly open the default logical file **OUTPUT.**

3. READ

The **READ** procedure assigns the value of components of a file to variables.

Syntax of **READ**: (default: file = **INPUT**)



The number of variables passed to the procedure determines the number of components read from the file. The components refer to the way the file is logically separated into individual data elements. Each component is of some data type which defines its size. Reading begins with the component pointed to by the file pointer. The first variable specified is assigned the value of this component and then the file pointer is advanced to the next component. This process is continued until all the variables specified are assigned values. The type of each variable must match the type of the file component being assigned to it.

TEXT FILES

If the file is of type **TEXT**, the variables can be of type **REAL**, **INTEGER**, subrange of **INTEGER**, **CHAR**, or strings. Strings are declared as single-dimensioned packed arrays of the type **CHAR.** These types can be intermixed as components of text files. Then they may be read by specifying variables which match in type and order, the components of the file.

> NOTE: The following characters have special meaning in a text file and may not be read as single characters. Use **FILE OF CHAR** to avoid this special processing:
> **HT = #Ø9 LF = #ØA CR = #ØD SUB = #1A**

If the variable is of type **CHAR**, then a single character is read from the file. If the variable is an array of **CHAR**, then the size of the array determines the number of characters read from the file. If an end of line or file mark is encountered before the array is full, then the characters read up to that point are left-justified in the array and the remaining elements are filled with blanks. Integer and real numbers are represented in files as strings of characters. Individual numbers in a file are separated by blanks or by an end of line mark. When a number is read, the character string representing the number is automatically converted to its real or integer value before being assigned to the variable. With text files, consecutive **READ** operations automatically skip end of line marks when reading integer, real, or boolean variables. When reading character or string variables, the end of line mark is not skipped. In this case, the procedure **READLN** must be executed to cause the file pointer to advance to the next line.

Example use with text files:

Consider the following file of data:

```
SAM JONES     25     183.5     369
MARY SMITH    23     105.4     356
        . . . . .
        . . . . .
```

and the declarations:

```
VAR name             : PACKED ARRAY[1..10] OF CHAR;
number, total        : INTEGER;
score                : REAL;
students             : TEST;
```

If the file pointer of "**students**" points to the beginning of a line (as it does immediately after a **RESET**) then:

```
READ(students,name,number,score,total)
```

would assign a string, integer, real, and integer value to the four specified variables. The file pointer would then point to the character immediately following the last value read.

NON-TEXT FILES

If the file is not of type **TEXT**, then all components of the file are of the same type. The components of a file may be declared to be of any type except the type **FILE** or structured types containing a component of type **FILE.** This means for example, that you could declare a file of records. Then an entire record can be read into a variable of the same record type. This however, requires that the file of records has been previously created through the use of the procedure **WRITE.** The reason for this is that all files which are not of type **TEXT** are read and written in binary form.

Example use with non-text files:

Assume the following declarations:

```
TYPE        food = RECORD
                   fruit     : (orange, grape, apple);
                   vegetable : (corn, okra, beans);
                   cost      : INTEGER;
                   END;

VAR         groceries : FILE OF food;
            item      : food;
```

then:

```
READ(groceries, item);
```

would assign one record from the file to the variable "item."

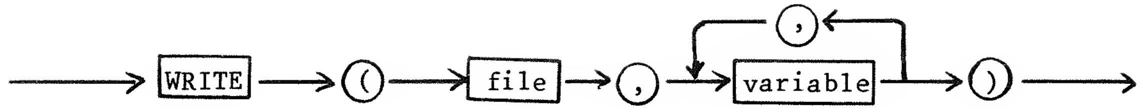
Care should be taken not to read past the end of a file. The function **EOF** is provided for preventing this from occurring. The program will not abort if you try to read past the end of file, but the value assigned to the variable will be some unknown value.

4. WRITE

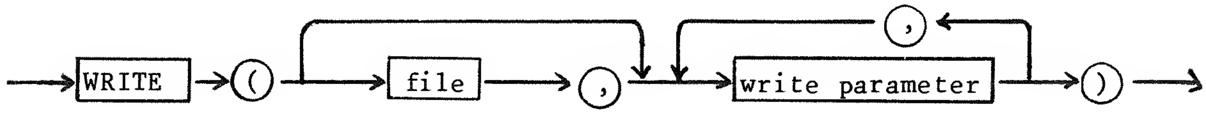
The procedure **WRITE** appends values to a file. The number of values passed to the procedure determines the number of values output to the file. If a file is declared as type **TEXT**, then output values can be specified as strings or expressions. If a file is declared as a type other then type **TEXT**, then output values are restricted to variables of the same type only.

Syntax of WRITE:

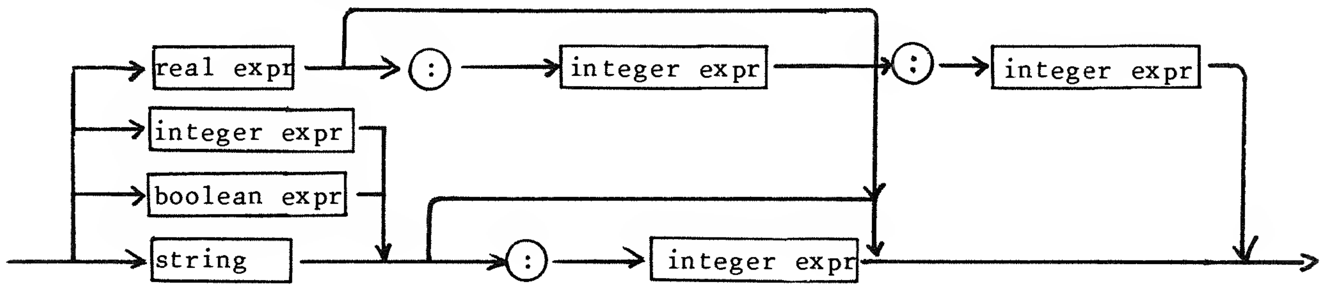
For non-text files:



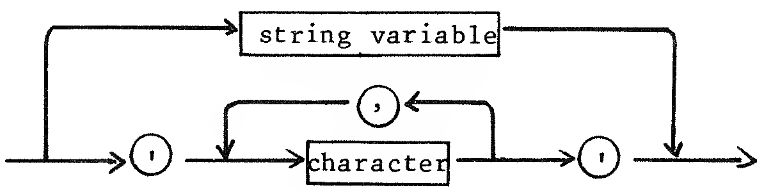
For text files: (default: file = **OUTPUT**)



Syntax of write parameter:



Syntax of string: (string variable = packed array of **CHAR**)



TEXT FILES

If the file is of type **TEXT**, then the values output to the file may be specified as strings or as boolean, integer, or real expressions. If a string is specified, then the characters of the string are output to the file. If a boolean expression is specified, then the characters forming either "TRUE" or "FALSE" are output to the file, depending on the value of the expression. If an integer or real expression is specified, then the value of the expression is converted to a character string before being output to the file. An integer expression may be output in hexadecimal or decimal base representation.

The number of characters to output for a value can be specified by an integer expression which follows the value, separated by a colon (:). If the number of characters is not specified for a particular value, then a default number of characters will be output.

For a string

If the number is less than the length of the string, then all the characters in the string are output. If the number is greater than the length of the string, then blanks will be appended to the string. The default number is the length of the string.

Example: **WRITE(' literal string' : 20)**

For a Boolean expression

The same rule applies for the strings 'FALSE' and 'TRUE'.

Example: **WRITE(a AND** b **: 10)**

For an Integer expression

If the number is less than the number of digits in the integer, then all the digits are output. If the number is greater than the number of digits, then the excess characters are output as blanks before the integer is output. The default number of digits for integers is 8. An integer value may be written in hexadecimal base format by specifying:

: width HEX

Example: **WRITE(outfile, n+5 :i, j :4 HEX)**

For a real expression

Two numbers may be specified for real values. The first number specifies the total field width. The second specifies the number of digits after the decimal point. If both are specified, the number will be written in fixed format. Otherwise, the number will be written in exponential format. The default field width for single precision is 12. The double precision default is 20. The maximum field width is 32.

Example: **WRITE(2.5*random :5, random/x:9:6)**

NON-TEXT FILES

If the file is not of type **TEXT**, then output values must be variables. Output directed to non-text files is in binary form. This means that values are output in the same form as they are stored. For example, an integer is not converted to a character string before it is output.

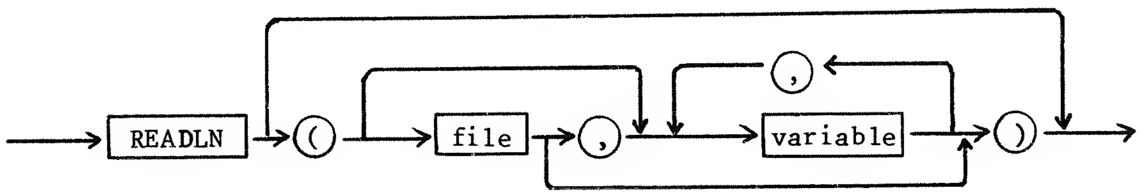
Example use with non-text files: **WRITE(groceries, item)**

5. READLN

This procedure can be used only with files of type **TEXT**. (See Chapter Four for a description of text files.)

The **READLN** procedure is similar to the **READ** procedure. The difference is that at the end of the read operation, the file pointer is advanced to the beginning of the next line.

Syntax of **READLN**: (default: file = **INPUT**)



The **READLN** procedure may be called without passing any variables to be read. When no variables are specified, then the procedure just advances the line pointer to the beginning of the next line.

The statement: **READLN(var1,var2,var3)**

is equivalent to: **BEGIN READ(var1,var2,var3); READLN END**

The function **EOLN** can be used to determine whether or not a file's pointer is at the end of a line.

Example use of **READLN**:

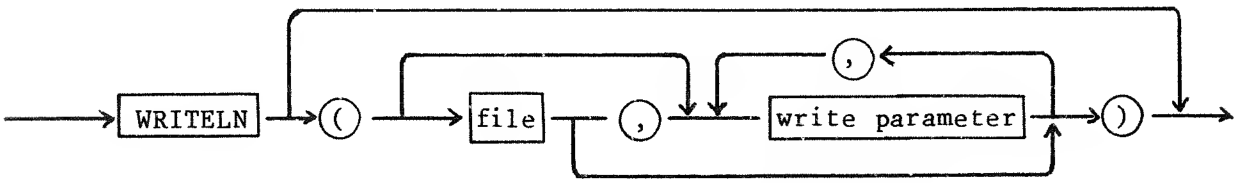
```
i := 0;
WHILE NOT EOF DO
   BEGIN
   i := i + 1;
   READLN(a[i]); (*reads one value from each line*)
   ........
   END;

WHILE NOT EOF (infile) DO
   BEGIN
   WHILE NOT EOLN(infile) DO
      BEGIN
      READ(infile,ch);
      .......
      END;
   READLN(infile);  (*advances file pointer to next line*)
   END;
```

6. WRITELN

This procedure can only be used with files of type **TEXT.** (See section 3.a. of Chapter Four for a description of text files.)

The **WRITELN** procedure is similar to the **WRITE** procedure. The difference is that at the end of the **WRITE** operation, an end-of-line mark is appended to the file.

Syntax of **WRITELN**:



(See **WRITE** for syntax of "write parameter.")

The **WRITELN** procedure may be called without passing any values to be written. When no values are specified, then the procedure just appends an end-of-line mark to the file.

The statement: **WRITELN(var1,var2,var3)**

is equivalent to: **BEGIN WRITE(var1,var2,var3); WRITELN END**

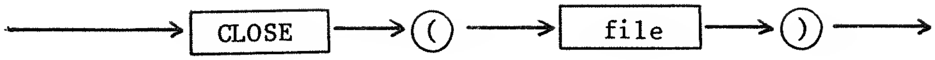
Example use of **WRITELN**:

```
(*writes 2 values on each line*)
FOR k := 1 TO 100 DO WRITELN(a[k],b[k]);

FOR j := 1 TO maximum DO
   BEGIN
   i := 0;
   REPEAT
      i := i+1;
      WRITE(number[j]);
   UNTIL (i=5) OR (number[j]>100);
   WRITELN;     (*advance file pointer to next line*)
   END;
```

7. CLOSE

The use of the **CLOSE** procedure will assure that file data will not be lost if the program abnormally terminates and does not properly close the file. The **CLOSE** procedure must be used with files which are components of structured variables. (See the appendix.)

Syntax of **CLOSE**:

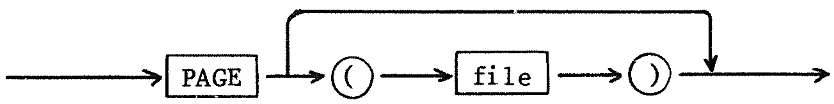


8. PAGE

The **PAGE** procedure appends a formfeed to a file. Formfeeds cause printers to skip to the top of the next page. This procedure provides a way of controlling the number of lines printed on the page.

This procedure may only be used with files of type **TEXT**.

Syntax of **PAGE**: (default: file = **OUTPUT**)



9. MESSAGE

The procedure **MESSAGE** may be used to output strings to the terminal (screen). It takes one parameter which is either a string constant or variable. A string constant is a sequence of characters enclosed in single quotation marks. A string variable is a variable declared as a packed array of characters.

Syntax of **MESSAGE**:

Programs which require only string output to the terminal can use this procedure rather than the **WRITE** procedure.

Example use of **MESSAGE**:

```
MESSAGE(' time to quit');
MESSAGE(string);
```

1. COMPILER OPTIONS

Compiler options are provided to change the behavior of the Pascal compiler. These options allow features to be enabled or disabled and can alter the code generated at compile time.

Compiler options are specified in comments. A comment that contains a dollar sign as the first character specifies an option. All compiler options have two states, on and off. An option is turned on by placing its name after the dollar sign. If the option name is preceded by the word "NO", then the option is turned off. Except where noted, the options may appear any place in a program.

1.a. DOUBLE

This option specifies that all real variables within the program should be double precision. This option must precede the **PROGRAM** statement. If it occurs anywhere else in the program, it will be ignored. If the option is off (the default), then real variables are single precision.

Example

```
(*$DOUBLE*)
PROGRAM DBL;
VAR
   R : REAL;
BEGIN
END.
```

In this program, the variable R will be declared as double precision.

1.b. FORDECL

This option is used to change the behavior of loop counters in **FOR** statements. If the option is turned on ("off" is the default), then all **FOR** loop counters are treated as temporary variables. They do not need to be declared, and even if a declaration is present, a new variable is used rather than the declared variable. These **FOR** loop counters are defined only within the loop, and disappear when the loop is exited.

Example:

```
PROGRAM FORLOOP;
(*$FORDECL*)
VAR
     A,I : INTEGER;
BEGIN
     A := 0;
     I := 0;
     FOR I := 0 to 4 do A := A + 1;
     WRITELN(OUTPUT,I,A);
END.
```

In the above program, the I used as a **FOR** loop counter is a different variable from the I declared in the **VAR** section. When the **WRITE** statement is executed, the values 0 and 10 will be printed.

1.c. INOUT

This option enables (makes available for use) the predeclared files **INPUT** and **OUTPUT**. The default status of this option is "on." If this option is turned off before the **PROGRAM** statement, then the files **INPUT** and **OUTPUT** will not be declared. Turning this option off prevents the reset of **INPUT** and the rewrite of **OUTPUT**, and can be used to avoid the prompts "**INPUT** =" and "**OUTPUT** =" when a program is run.

Example:

```
(*$NO INOUT*)
PROGRAM NOPROMPTS;
BEGIN
     MESSAGE('I WAS NOT PROMPTED FOR INPUT AND OUTPUT')
END.
```

1.d. IF

The **IF** option provides conditional compilation. The word **IF** is followed by the name of a boolean constant. If the constant has the value "TRUE," then compilation continues as if the option had not been present. If the constant has the value "FALSE," then compilation stops at that point, and all text is treated as comments until a **(*$NO IF*)** is encountered. Note that **IF** options do not nest. That is, an **IF** option should not occur within the scope of another **IF** option. The **IF** option can be used to configure a program for different environments with minimum changes to the source. It is also useful for removing debugging statements once the program is working properly.

Example use of **IF** option:

```
PROGRAM Test;
CONST
     debug = false;

FUNCTION FACTORIAL(I : INTEGER) : REAL;
BEGIN
     IF I = 0 THEN FACTORIAL := 1
     ELSE BEGIN
          (*$IF DEBUG*)
          WRITELN(OUTPUT,'CALLING FACTORIAL(',I-1,')');
          (*$NO IF*)
          FACTORIAL := I * FACTORIAL(I-1);
          END;
END;  (* FACTORIAL *)

BEGIN
     WRITELN(OUTPUT,'FACTORIAL(20)=',FACTORIAL(20));
END.
```

In the above program, the **WRITE** statement within the recursive function **FACTORIAL** could be turned on during debugging by setting debug to TRUE. Once the program is running, it can be recompiled with debug set to FALSE. The **WRITE** statement will be effectively removed. In fact, since no code is generated for it, the resulting object program will be shorter. This has the same effect as removing the statement with an editor or placing open and close comments around it. The advantage is that many statements can be disabled or enabled with a single change to the source code. Also, it is simple to reenable debugging statements should it become necessary in the future.

1.e. NULLBODY

The **nullbody** option is used to disable code generation for a procedure, function or program. The **NULLBODY** option should occur after the **BEGIN** that starts the block and before any executable statements. **NULLBODY** will prevent code from being generated and can be used when procedures are being compiled separately. Since every program must have a **PROGRAM** statement and a main program body, it is necessary to use **NULLBODY** to disable code generation for the main program when a subroutine library is being compiled.

Example use of **NULLBODY**:

```
PROGRAM SUBLIBRARY;
TYPE
   STRING = PACKED ARRAY[1..80] OF CHAR;

PROCEDURE CONCATENATE (VAR S1, S2, RESULT: STRING);
BEGIN
 (* BODY OF CONCATENATE *)
END;

PROCEDURE MID$ (VAR S : STRING; FIRST, LAST : INTEGER;
           VAR RESULT : STRING);

BEGIN
 (* BODY OF MID$ *)
END;

BEGIN
 (*$NULLBODY*)
END.
```

If the above program is compiled, the object file will contain code only for the two procedures: **CONCATENATE** and **MID$**. There will be no main program. This allows these procedures to be linked to another program.

1.f. INCLUDE

The **INCLUDE** option is used to specify within a program, the name of a file which contains Pascal statements which you want included in the compilation process. When the compiler encounters an **INCLUDE** option, it opens the specified file and compiles all the Pascal source code in the file before continuing compilation of the current file. The **INCLUDE** option allows you to include commonly used routines or declarations in a program without actually having the code present. You simply tell the compiler the name of the file containing the Pascal statements and it will include those statements as it compiles.

The **INCLUDE** options may be nested. That is, you may include a file which also contains an **INCLUDE** compiler option. There is no limit to the number of nested **INCLUDE**s. However, the compiler must maintain a file descriptor for each file that is open at any given time. The file descriptors are allocated memory from the heap. If too many files are open at a time, the compiler may run out of heap during the compile process.

Example use of the **INCLUDE** option:

```
PROGRAM sample;
{DECLARE contains the declarations for this program}
(*$INCLUDE 'DECLARE'*)  {note the quotes}
BEGIN
  {BODY contains the statement body for this program}
  (*$INCLUDE 'BODY'*)
END.
```

1.g. LIST

The **LIST** option allows you to turn the compiler listing on and off within a program. "On" is the default. Therefore, the compiler will by default generate a listing which contains all the lines of a program. If it is desired to discard some of the lines of a program from the compiler generated listing, **(*$NO LIST*)** may be used to tell the compiler to discard all subsequent lines of the program from the listing. The compiler does not stop compiling subsequent lines, it just does not output them to the listing. Object code is still generated. If you wish to turn the compiler listing back on, then, **(*$LIST*)** tells the compiler to start outputting all subsequent lines to the listing again.

The **LIST** option may be useful when compiling frequently used routines which you know will compile correctly. It provides a method to shorten compiler listings, saving paper when printing, and making it easier to locate other procedures or functions by uncluttering lengthy program listings.

Example using the compiler **LIST** option:

```
PROGRAM sample;
VAR ...
  PROCEDURE useoften;
  (*$NO LIST*)      {turn off listing for useoften}
  VAR ...
  BEGIN
   ...
  END;    {end of procedure useoften}
(*$LIST*) {turn listing back on for program}
BEGIN     {beginning of main program}
  ...
END.
```

1.h. PAGESIZE

The listing generated by the compiler has printer control information (formfeed between each page. The compiler outputs a formfeed (hex 0C) to the listing every 62 lines. The formfeed causes most printers to advance the paper to the top of the next page. The **PAGESIZE** option allows you to change the number of lines that the compiler will output to the listing between formfeeds. The actual number of lines output between formfeeds is 2 more than the number specified by the **PAGESIZE** option. This is to allow for the heading.

Most operating systems control paging when outputting data to a line printer. The operating system itself maintains a line counter and outputs a formfeed to the line printer after so many lines have been sent to the printer. A command is typically provided to set the number of lines per page or to turn paging control off entirely. If the operating system is controlling paging, the listing generated by the compiler may not be paged properly (i.e., the compiler heading may not appear at the top of each page). The number of lines per page used by the operating system should be equal to the number of lines per page used by the compiler, or the operating system paging must be turned off, if compiler-generated listings are to be printed properly.

Example use of the compiler **PAGE** option:

```
(*$PAGESIZE 50*)             {set the number of lines/page to 50}
PROGRAM sample;
{the operating system paging should be set to 52
 or be turned off entirely}
...
BEGIN
  ...
END.
```

1.i. WIDELIST

The compiler now generates line numbers for each line of a listing. The **WIDELIST** option is used to specify that you want the compiler to additionally generate hexadecimal addresses which show the location of the object code for a particular line relative to the start of the procedure, function, or program in which the line appears. This information is useful when used in conjunction with the linking loader to determine the location within a program of a fatal error. You may use the **S** command of the linking loader to display the starting address of each routine loaded. Then use the **R** command to run the program. When the program terminates with a fatal error, the absolute hex address of the error is displayed. You may use this address along with the addresses displayed by the **S** command to determine in which routine the error occurred. By subtracting the address of the error from the starting address of the routine in which the error occurred, you obtain the relative address of the error within that routine. This address corresponds to the address printed on the listing.

Example use of the compiler **WIDELIST** option:

```
(*$WIDELIST*)    {tell the compiler to print hex addresses}
PROGRAM sample;
...
BEGIN
  ...
END.
```

1.j. RANGECHK

A common error which occurs in programs which utilize arrays is to index the array with a value which is outside the array bounds (for example, an array with bounds 1..10 is indexed with the value 11). A common error in programs which utilize subranges is to assign a value which is outside the subrange (for example, a variable is declared as type 0..225 and is assigned the value 275). A common error in programs which utilize enumerations is to increment or decrement past the first or last value of the enumeration [for example, **SUCC(color)** is executed when **color** is equal to **blue** and **color** is of type **(red, green, blue)**].

All of these errors may be trapped, causing an appropriate runtime error message to be displayed when such an error occurs during program execution. The **RANGECHK** option tells the compiler to generate extra code to detect and report errors of the above kind when the compiled program is executed.

Since the **RANGECHK** option does cause additional object code to be generated, you should generally use it only during the debugging stage of program development. The **RANGECHK** option may be turned on and off throughout a program. The default is "off". The **IF** compiler option may be used to conditionally turn the **RANGECHK** option on and off as needed for debugging purposes.

Example use of the compiler **RANGECHK** option.

```
PROGRAM sample;
VAR    A,B : ARRAY [1..200] OF CHAR;
       J,K : INTEGER;
       ...
BEGIN
  WRITE(OUTPUT,'Enter size of array: ');
  (*$RANGECHK*)                {turn range checking on}
  ...
  FOR K :=1 TO J DO A[K] := B[K+1];
  (*$NO RANGECHK*)             {turn range checking off}
  ...
END.
```

NOTE: The **RANGECHK** option will not detect an error on subrange variables which are assigned invalid values via a **READ** statement. To trap these errors, you must assign the read-in value to a subrange variable.

```
READ(VALUE);
SUBRANGE_VARIABLE := VALUE;
```

1.k. PTRCHECK

A common error which occurs in programs which utilize dynamic pointer variables is the inadvertent assignment of the value **NIL** to a pointer and then the subsequent attempt to use the value pointed to in an expression or in an assignment to a static variable. Another common error is the attempt to utilize an uninitialized pointer. An uninitialized pointer may not point to a location within the allocated heap of the program. It may point into the executing code of the program, making it possible to write data over the instructions, causing very unpredictable results.

The **PTRCHECK** option is used to tell the compiler to generate extra code in the compiled program to detect and report either of the above types of errors when the program executes. This extra code causes the program to terminate and display an appropriate error message when an invalid use of a pointer variable is detected. The **PTRCHECK** option may be turned on and off throughout a program. The default is "off".

Example use of the compiler **PTRCHECK** option:

```
PROGRAM sample;
TYPE  customer = RECORD name,add : ARRAY[1..9] OF CHAR END;
VAR   cust   : ^customer;
BEGIN
  (*$PTRCHECK*)
  ...
  WHILE cust<>NIL DO
END.
```

2. ERROR MESSAGES

2.a. Compiler Error Codes

Below is a list of the error codes that may be generated by the compiler, along with a brief explanation of their meanings.

2 IDENTIFIER EXPECTED
3 'PROGRAM' EXPECTED
4 ')' EXPECTED
5 ':' EXPECTED
6 ILLEGAL SYMBOL
8 'OF' EXPECTED
9 '(' EXPECTED
10 ERROR IN TYPE
11 LEFT BRACKET '[' OR '(.' EXPECTED
12 RIGHT BRACKET ']' OR '.)' EXPECTED
13 'END' EXPECTED
14 ';' EXPECTED
15 INTEGER EXPECTED
16 '=' EXPECTED
17 'BEGIN' EXPECTED
20 ',' EXPECTED
22 '..' EXPECTED
23 '.' EXPECTED
49 'ARRAY' EXPECTED
50 CONSTANT EXPECTED
51 ':=' EXPECTED
52 'THEN' EXPECTED
53 'UNTIL' EXPECTED
54 'DO' EXPECTED
55 'TO'/'DOWNTO' EXPECTED
57 'FILE' EXPECTED
58 INVALID OR MISSING OPERAND IN AN EXPRESSION
62 DECIMAL PLACE ALLOWED ONLY FOR REAL
66 TYPE IDENTIFIER EXPECTED
80 OPEN COMMENT WITHIN A COMMENT
81 UNKNOWN OPTION
82 # REQUIRES A 2 CHARACTER HEX VALUE OR ##

101 IDENTIFIER DECLARED TWICE
102 LOWER BOUND EXCEEDS UPPER BOUND
103 IDENTIFIER IS NOT OF APPROPRIATE CLASS
104 UNDECLARED IDENFITIER
105 CLASS OF IDENTIFIER IS NOT VARIABLE
107 INCOMPATIBLE SUBRANGE TYPES
113 ARRAY BOUNDS MUST BE SCALAR
117 UNSATISFIED FORWARD REFERENCE TO A TYPE IDENTIFIER OF A POINTER
119 ';' EXPECTED (PARAMETER LIST NOT ALLOWED)
120 FUNCTION RESULT MUST BE SCALAR, SUBRANGE, OR POINTER
123 FUNCTION RESULT EXPECTED
126 IMPROPER NUMBER OF PARAMETERS
127 TYPE OF ACTUAL PARAMETER DOES NOT MATCH FORMAL PARAMETER
129 TYPE CONFLICT OF OPERANDS IN AN EXPRESSION
132 COMPARISON WITH '>' OR '<' NOT ALLOWED ON SETS
134 ILLEGAL TYPE OF OPERANDS
135 TYPE OF EXPRESSION MUST BE BOOLEAN
136 SET ELEMENT TYPE MUST BE SOME ENUMERATION TYPE
138 TYPE OF VARIABLE IS NOT ARRAY
140 TYPE OF VARIABLE IS NOT RECORD
141 TYPE OF VARIABLE IS NOT POINTER
148 SET BOUNDS OUT OF RANGE
152 NO SUCH FIELD IN THIS RECORD
154 ACTUAL PARAMETER MUST BE A VARIABLE
156 MULTIDEFINED CASE LABEL
161 PROCEDURE OR FUNCTION ALREADY DECLARED AT A PREVIOUS LEVEL
165 LABEL ALREADY DEFINED
167 UNDECLARED LABEL
168 LABEL NOT DEFINED
182 "FOR" EXPRESSION MUST BE OF SOME ENUMERATION TYPE
183 "CASE" EXPRESSION MUST BE OF SOME ENUMERATION TYPE
184 "FOR" VARIABLE MUST BE LOCAL
185 OPERATION DEFINED FOR TEXT ONLY
186 OPERATION NOT DEFINED FOR TEXT FILES
193 ACCESS STATEMENT MISSING FOR COMMON
199 FEATURE NOT IMPLEMENTED
202 STRING CONSTANT CANNOT SPAN LINES
203 INTEGER CONSTANT TOO LARGE
210 FIELD WIDTH MUST BE INTEGER
211 FRACTION LENGTH MUST BE OF TYPE INTEGER
212 HEX FORMAT ALLOWED ONLY FOR TYPE INTEGER
219 PARAMETER MUST BE OF TYPE FILE
220 PARAMETER MUST BE OF TYPE INTEGER
223 PARAMETER MUST BE OF TYPE POINTER
230 ILLEGAL TYPE OF PARAMETER IN STANDARD PROCEDURE CALL
250 TOO MANY NESTED SCOPES - LIMIT IS 15
401 OPEN COMMENT ENCOUNTERED IN A COMMENT
403 TOO MANY PROCEDURE NESTING LEVELS
404 ARRAY BOUNDS MUST BE SCALAR

2.b. Runtime Error Codes

Below is a list of the error messages that can be generated during execution of a Pascal program, along with their causes and solutions.

01) OUT OF STACK

cause: insufficient amount of stack available
cure : If compiling
with **PASCAL** : switch to **PASCALB**
with **PASCALB** : specify more stack space
PASCALB <stack> file
If executing
with **RUN** : specify more stack space
RUN file stack
with **/CMD** : specify more stack space when using **B** command of **LINKLOAD**

02) OUT OF HEAP

cause: insufficient amount of heap available
cure : If compiling
with **PASCAL** : switch to **PASCALB**
with **PASCALB** : specify less stack space
If executing
with **RUN** : specify less stack space
with **/CMD** : specify less stack space when using **B** command of **LINKLOAD**

03) BAD POINTER

cause: damaged object file or error in program which causes executing code to be overwritten with data
cure: If executing one of the system **/CMD** files:
restore defective **/CMD** file from the original master disk
If executing a user written program:
recompile the program using the **RANGECHK** and **PTRCHECK** options and execute once again. Invalid array indexing and most invalid pointer referencing will be trapped. If a range or pointer error message is displayed, locate and fix the programming error.

04) BAD LEVEL

see error 03

05) DIVIDE BY 0

cause: an integer or real divide operation with a divisor of 0
cure : prevent divisor from becoming 0

06) UNDEFINED PCODE

see error 03

07) INVALID SET

cause: set operation results in set with more than 256 members
cure : restrict set operations to 256 member sets

08) BAD RUNTIME CALL
see error 03

09) IO ERROR
cause: 1 - file does not exist
2 - disk is full
3 - bad disk or hardware
cure : 1 - specify correct file name
2 - clear some space on the disk
3 - run diagnostics

0A) SET ELEMENT TOO LARGE
cause: attempt to assign an ordinal value > 256 to a set
cure : limit sets to 256 members

EB) ATTEMPT TO WRITE TO INPUT FILE
cause: opening an output file using **RESET**
cure : open the output file using **REWRITE**

EC) FILE NOT OPEN
cause: attempt to read or write an unopened file
cure : open the file using **RESET** or **REWRITE**

ED) ATTEMPT TO READ OUTPUT FILE
cause: opening an input file using **REWRITE**
cure : open the input file using **RESET**

EE) NO MEMORY FOR FILE BUFFER
cause: not enough space for file buffer in heap
cure : execute program using less stack

10) RANGE CHECK
cause: invalid array index, subrange value, or enumeration value
cure : correct invalid array indexing and/or invalid values

11) BAD DIGIT IN NUMBER
cause: attempt to **READ** or **DECODE** an invalid number
cure : make sure all numbers read or decoded are legal numbers

12) PUT ERROR
cause: attempt to output an undefined file buffer variable
cure : assign a proper value to the file buffer variable

13) OVERFLOW
cause: a real arithmetic calculation overflows
cure : limit real numbers to the maximum size

15) UNDERFLOW
cause: a real divide operation causes underflow
cure : limit real numbers to the minimum non-zero size

16) LOG NEGATIVE
cause: attempt to take the natural log of a number <= 0
cure : log is valid positive numbers only

17) **SORT, X^Y NEGATIVE**

cause: attempt to take the square root of a negative number or
attempt to raise a negative number to a real power

cure : square root is valid only for number >= 0
only positive numbers may be raised to a real power

3. STANDARD 7-BIT USASCII CHARACTER SET

Decimal	Octal	Hex	Graphic	Name
0.	000	00	@	NUL (used for padding) <null>
1.	001	01	^A	SOH (start of header)
2.	002	02	^B	STX (start of text)
3.	003	03	^C	ETX (end of text)
4.	004	04	^D	EOT (end of transmission)
5.	005	05	^E	ENQ (enquiry)
6.	006	06	^F	ACK (acknowledge)
7.	007	07	^G	BEL (bell or alarm)
8.	010	08	^H	BS (backspace) <bs>
9.	011	09	^I	HT (horizontal tab) <tab>
10.	012	0A	^J	LF (line feed) <lf>
11.	013	0B	^K	VT (vertical tab)
12.	014	0C	^L	FF (form feed, new page) <ff>
13.	015	0D	^M	CR (carriage return) <cr>
14.	016	0E	^N	SO (shift out)
15.	017	0F	^O	SI (shift in)
16.	020	10	^P	DLE (data link escape)
17.	021	11	^Q	DC1 (device control 1, XON)
18.	022	12	^R	DC2 (device control 2)
19.	023	13	^S	DC3 (device control 3, XOFF)
20.	024	14	^T	DC4 (device control 4)
21.	025	15	^U	NAK (negative acknowledge)
22.	026	16	^V	SYN (synchronous idle)
23.	027	17	^W	ETB (end transmission block)
24.	030	18	^X	CAN (cancel)
25.	031	19	^Y	EM (end of medium)
26.	032	1A	^Z	SUB (substitute)
27.	033	1B	^[	ESCAPE (alter mode, SEL) <esc>
28.	034	1C	^\	FS (file separator)
29.	035	1D	^]	GS (group separator)
30.	036	1E	^^	RS (record separator)
31.	037	1F	^	US (unit separator)
32.	040	20	_	space or blank <sp>
33.	041	21	!	exclamation mark
34.	042	22	"	double quote
35.	043	23	#	number sign (hash mark)
36.	044	24	$	dollar sign
37.	045	25	%	percent sign
38.	046	26	&	ampersand sign
39.	047	27	'	single quote (apostrophe)
40.	050	28	(	left parenthesis
41.	051	29	)	right parenthesis
42.	052	2A	*	asterisk (star)
43.	053	2B	+	plus sign
44.	054	2C	,	comma
45.	055	2D	-	minus sign (dash)
46.	056	2E	.	period (decimal point)
47.	057	2F	/	(right) slash
48.	060	30	0	numeral zero
49.	061	31	1	numeral one

50.	062	32	2	numeral two
51.	063	33	3	numeral three
52.	064	34	4	numeral four
53.	065	35	5	numeral five
54.	066	36	6	numeral six
55.	067	37	7	numeral seven
56.	070	38	8	numeral eight
57.	071	39	9	numeral nine
58.	072	3A	:	colon
59.	073	3B	;	semi-colon
60.	074	3C	<	less-than sign
61.	075	3D	=	equal sign
62.	076	3E	>	greater-than sign
63.	077	3F	?	question mark
64.	100	40	@	at sign
65.	101	41	A	upper-case letter ABLE
66.	102	42	B	upper-case letter BAKER
67.	103	43	C	upper-case letter CHARLIE
68.	104	44	D	upper-case letter DELTA
69.	105	45	E	upper-case letter ECHO
70.	106	46	F	upper-case letter FOXTROT
71.	107	47	G	upper-case letter GOLF
72.	110	48	H	upper-case letter HOTEL
73.	111	49	I	upper-case letter INDIA
74.	112	4A	J	upper-case letter JERICHO
75.	113	4B	K	upper-case letter KAPPA
76.	114	4C	L	upper-case letter LIMA
77.	115	4D	M	upper-case letter MIKE
78.	116	4E	N	upper-case letter NOVEMBER
79.	117	4F	O	upper-case letter OSCAR
80.	120	50	P	upper-case letter PAPPA
81.	121	51	Q	upper-case letter QUEBEC
82.	122	52	R	upper-case letter ROMEO
83.	123	53	S	upper-case letter SIERRA
84.	124	54	T	upper-case letter TANGO
85.	125	55	U	upper-case letter UNICORN
86.	126	56	V	upper-case letter VICTOR
87.	127	57	W	upper-case letter WHISKY
88.	130	58	X	upper-case letter XRAY
89.	131	59	Y	upper-case letter YANKEE
90.	132	5A	Z	upper-case letter ZEBRA
91.	133	5B	[	left square bracket
92.	134	5C	\	left slash (backslash)
93.	135	5D	]	right square bracket
94.	136	5E	^	up arrow (caret)
95.	137	5F	_	underscore
96.	140	60	`	(single) back quote
97.	141	61	a	lower-case letter able
98.	142	62	b	lower-case letter baker
99.	143	63	c	lower-case letter charlie
100.	144	64	d	lower-case letter delta
101.	145	65	e	lower-case letter echo
102.	146	66	f	lower-case letter foxtrot
103.	147	67	g	lower-case letter golf
104.	150	68	h	lower-case letter hotel

105.	151	69	i	lower-case letter india
106.	152	6A	j	lower-case letter jericho
107.	153	6B	k	lower-case letter kappa
108.	154	6C	l	lower-case letter lima
109.	155	6D	m	lower-case letter mike
110.	156	6E	n	lower-case letter november
111.	157	6F	o	lower-case letter oscar
112.	160	70	p	lower-case letter pappa
113.	161	71	q	lower-case letter quebec
114.	162	72	r	lower-case letter romeo
115.	163	73	s	lower-case letter sierra
116.	164	74	t	lower-case letter tango
117.	165	75	u	lower-case letter unicorn
118.	166	76	v	lower-case letter victor
119.	167	77	w	lower-case letter whisky
120.	170	78	x	lower-case letter xray
121.	171	79	y	lower-case letter yankee
122.	172	7A	z	lower-case letter zebra
123.	173	7B	{	left curly brace
124.	174	7C	|	vertical bar
125.	175	7D	}	right curly brace
126.	176	7E	~	tilde
127.	177	7F	<rubout>	DEL <del>

4. DIFFERENCES FROM STANDARD

The standard used is defined by "User Manual and Report", second section, Jensen and Wirth, Springer-Verlag. The following sections pertain to the differences in Alcor Systems implementation of Pascal as compared to the standard. The extensions are added to provide extra power to the language. All implementations of Pascal by Alcor systems contain these added features. If a program is to be transported to a computer system using some other implementation of Pascal, these features should not be used in the program.

4.a. Omissions

1) Procedures or functions may not be passed as parameters to other procedures or functions.

4.b. Extensions

1) Common variables which provide a mechanism for statically allocating local variables are implemented through the use of two new declaration parts: **COMMON** and **ACCESS**.

2) The declaration sections **LABEL**, **CONST**, **TYPE**, **VAR**, **COMMON**, and **ACCESS** may appear any number of times and in any order within a block.

3) The Type Transfer Operator allows variables to be referenced through the use of a type template.

4) Single elements of packed structures may be passed as parameters.

5) The **OTHERWISE** clause is implemented in the **CASE** statement. If omitted, and there is no match, execution transfers to the next statement.

6) Identifiers can include the characters '_' and '$'. Also, no distinction is made between upper and lower case letters.

7) Integer constants or characters may be represented in hex.

8) Mixed mode arithmetic is implemented.

9) The procedures **READ** and **READLN** will accept string and boolean variables.

10) External procedures or functions may be declared. This feature provides a way of accessing external routines.

11) Input files are not opened until necessary. This eliminates the synchronization problem when doing interactive input from a terminal.

12) Labels may range from -32768 to 32767.

13) Alternate symbols are implemented for brackets and the pointer symbol.

14) The **LOCATION** function allows the determination of the address of a variable of a procedure.

15) The **SIZE** function allows the size of a type to be determined.

16) The **HB** functions returns the high byte of an integer variable.

17) The **LB** function returns the low byte of an integer variable.

18) The procedure **MESSAGE** provides an additional method for handling string output to a terminal.

19) The procedure **CLOSE** allows files to be explicitly closed.

20) The procedure **ESCAPE** allows exiting a block at any point within the block.

21) The type **STRING** is a predefined dynamic data type. A string function library is provided for use with this data type.

22) Libraries are provided to access the hardware features of the specific machine.

23) Compiler options are provided to control various functions.

4.c. Other Implementation Characteristics

The following is a list of specific implementation decisions which are not defined by the standard.

1) Only the first 8 characters of an identifier are stored. This means that identifier names should be selected such that the first 8 characters form a unique name.

2) There is a limit of 256 elements for sets, enumerations, **CASE** statement labels, and parameters to a procedure or function.

3) Pascal source code is restricted to 80 columns.

4) The association of logical files to physical devices is made either interactively from the terminal or through a procedure call.

The following is a list of characteristics which are slightly altered from the standard.

1) Operator precedence has been altered to eliminate the need for exces sive use of parentheses in expressions. The precedence is the same as that used in BASIC. The difference is the precedence assigned to the Boolean operators. The precedence defined by the standard makes the Boolean operator **OR** equal in precedence with + and -, the Boolean operator **AND** equal in precedence with *, /, **DIV**, and **MOD**, and **NOT** has the highest precedence of any operator except the parentheses. Parentheses may be used when transportable programs are being written to maintain compatability with the standard. This alteration of precedence should not cause any problems when transferring programs written in standard Pascal to Alcor Pascal.

2) Although structured variables may contain components of type **FILE**, the I/O routines will accept only simple variable names. Therefore, files within structured variables may be used only in a restricted manner.

3) A **GOTO** statement may not reference a label outside the block in which the statement appears.

5. THE TYPE STRING

The standard Pascal string is defined to be a **PACKED ARRAY OF CHAR.** Variables of this type are restricted to a predetermined size. (That is, the size of the array must be specified and cannot be altered during program execution). In contrast, the predefined type **STRING** is dynamic. The size of a variable declared as type **STRING** is determined during program execution. Variables of this type may change in size as the program executes. In addition, variables of type **STRING** may be used in conjunction with a runtime library of string manipulation functions.

Syntax of type **STRING**:

Example:

```
VAR  strl, str2, str3 : STRING;
```

5.a. Assigning Values to Dynamic String Variables

A dynamic string may be created through the use of the predeclared transfer function **BLDSTR.** This function has one parameter which may be either a variable of the type **PACKED ARRAY OF CHAR** or a string constant. The function returns a dynamic string of the same length as the array or string constant passed to it.

Example:

```
strl := BLDSTR ('literal string constant');

str2 := BLDSTR (stringconstant);

str3 := BLDSTR (arrayvariable);
```

The procedures **READ** and **READLN** have been extended to accept variables of the type **STRING.** When a variable of type **STRING** is specified, all characters from the current file pointer to the end of line mark are read. The size of the string is then equal to the number of characters read. If a read is performed while at the end of line mark, the string variable is assigned an empty string. An empty string is a string of zero length.

Example:

```
READ (strl);

READLN (filename, str2);
```

A string variable may be assigned to another string variable. An assignment between two string variables results in both string variables referencing the same string. (That is, both string variables point to the same location in memory.)

Example:

```
strl := str2;
```

NOTE: For most applications, the preferred method of assignment between two string variables is through the use of the library function **CPYSTRING.** If two string variables point to the same location and one is disposed (using **DISPOSE**), then both string variables will become undefined.

A string variable may have assigned to it a string formed by one of the string manipulation functions in the runtime library. For example, there is a function provided which may be used for assignment between two string variables. The function **CPYSTR** takes a string variable as a parameter and copies it to another location. The string appearing on the left side of the equal sign then references the new location. In other words, instead of having one copy of the string as in the above example, there are now two copies.

Example:

```
strl := CPYSTR (str2);
```

5.b. Outputting Dynamic String Variables

The **WRITE** and **WRITELN** procedures have been extended to accept variables of the type **STRING.** When a dynamic string is output, the number of characters written is equal to the length of the string.

5.c. Converting a Dynamic String Into an Array

Dynamic strings can only be accessed as a whole. (That is, the individual characters of the string cannot be accessed). The predeclared procedure **GETSTR** will copy a dynamic string variable into a variable of the type **PACKED ARRAY OF CHAR.** It accepts two parameters. The first parameter is the dynamic string variable. The second parameter is the array variable. The string is left-justified in the array. If the string is longer than the array, then it is truncated. If the string is shorter than the array, then the array is padded with blanks.

Example:

```
GETSTR (strl, arrayvariable);
```

5.d. Recovering Memory Used by a Dynamic String

The memory used by a dynamic string may be recovered through the use of the standard procedure **DISPOSE.** When a string variable is passed to the **DISPOSE** procedure, the memory used by the string is freed and the string variable becomes undefined. In addition, any other string variable which points to the same string will become undefined. Each time a string variable is assigned a value, it points to a new string and the old string is then lost. The memory it uses cannot be recovered. Therefore, before assigning the string variable a new value, the memory used by the old value should be recovered if space is important.

Example:

```
strl := BLDSTR ('this is the first value');
...
DISPOSE (strl);
strl := BLDSTR ('this is the second');
```

5.e. Using the String Library

There is a long list of string manipulation functions available in the runtime library. In order for a program to have access to these functions, it must include an external declaration for each function used. A file of external declarations for all the string functions is supplied on disk. You can use the Model III Pascal editor program to insert this file into the programs that use these functions. The declarations for any functions which are not used may be deleted if desired. If only one or two functions are used, you may prefer to type in the external declarations. The NETWORK PASCAL SYSTEM REFERENCE GUIDE (page E-25) lists these functions.

Example use of dynamic strings:

```
PROGRAM sample;

VAR    firstname, lastname,
       space, fullname        : STRING;

FUNCTION CONC(s1,s2 : STRING) : STRING; EXTERNAL;
(*CONC is a string library function which concatenates 2 strings*)

BEGIN
       space := BLDSTR (' ');
       WRITELN (' enter first name');
       READLN (firstname);
       WRITELN (' enter last name');
       READLN (last name);
       fullname := CONC (CONC(firstname, space),lastname);
       WRITELN (fullname)
END.
```

6. I/O PROCEDURES GET AND PUT

File buffer variables and the procedures **GET** and **PUT** are I/O features of Pascal which are not often used. The procedures **READ** and **WRITE** are abbreviated forms for accomplishing the same I/O tasks. However, file buffer variables do provide a means of performing "lookahead" in a file. (That is, you may check the value of the next component in a file before actually reading it.) The ability to perform "lookahead" may offer some advantages in certain applications (for example, the scanner of a compiler).

6.a. File Buffer Variables

There is a file buffer variable associated with each file in a program. The buffer variable is used as temporary storage for file components as they are passed to or from the associated file. The buffer variable is the same size and type as an individual component of the file. The individual components of **TEXT** files are characters. Therefore, the file buffer variable associated with a file of type **TEXT** has a size of one byte (8 bits) and is of type **CHAR.** A file declared as **FILE OF INTEGER** consists of individual components of type **INTEGER.** The associated file buffer variable will have a size of two bytes (that is, integers require two bytes of storage) and will be of type **INTEGER.**

The buffer variable associated with a particular file may be referenced in the same manner as pointer variables, using either the ^ or **@** symbol. The buffer variable of a particular file is referenced by following the logical file name with either of these two symbols. For example, the buffer variable of the logical file **INPUT** is referenced by either **INPUT^** or **INPUT@.**

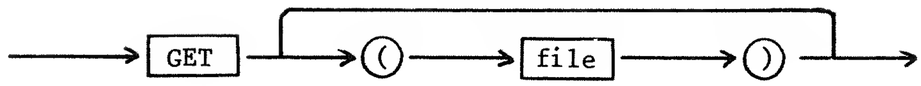
Files are opened for reading or writing by the procedures **RESET** or **REWRITE**, respectively. When a file is opened by **RESET**, the buffer variable associated with the file is assigned the value of the first component in the file. If the file is empty at the time it is opened, then the value of the buffer variable is undefined. When a file is opened by **REWRITE**, its associated buffer variable is undefined.

File buffer variables may be assigned values using the assignment statement. For example, **OUTPUT@ := 'A'** will assign the character **A** to the file buffer associated with the logical file **OUTPUT.** Additionally, the procedures **READ**, **READLN**, and **GET** will alter values of file buffer variables associated with input files. The buffer variables associated with output files become undefined after performing the operation specified by the **WRITE**, **WRITELN**, or **PUT** procedures.

6.b. The GET Procedure

The **GET** procedure assigns the value of the next component of a file to the buffer variable associated with that file. If there is no next component (that is, the end of the file has been reached), then **EOF** on that file becomes TRUE and the value of the buffer variable is undefined.

Syntax of **GET**: (default: file = **INPUT**)



Examples:

GET {assigns the next character of the logical file **INPUT** to the buffer variable **INPUT@**}

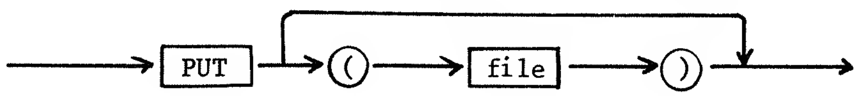
GET(F) {assigns the next component of the logical file **F** to the buffer variable **F@**}

READ(f,x) {is equivalent to **x** := **f@; GET(f)**}

6.c. The PUT Procedure

The **PUT** procedure appends the value of the buffer variable for a particular file to the end of that file. After the operation, the value of the buffer variable becomes undefined.

Syntax of **PUT**: (default: file = **OUTPUT**)



Examples:

PUT {appends value of the buffer variable **OUTPUT@** to the end of the logical file **OUTPUT**}

PUT(F) {appends the value of the buffer variable **F@** to the end of the logical file **F**}

WRITE(f,x) {is equivalent to **f@:=x; PUT(f)**}

Example use of **GET** and **PUT** and file buffer variables:

The following program copies the contents of logical file **infile** to logical file **outfile.**

```
PROGRAM  filecopy;
VAR      infile, outfile  : TEXT;
BEGIN
   RESET(infile);                {infile@ = first character}
   REWRITE(outfile);             {outfile@ is undefined}
   WHILE NOT EOF(infile) DO
      BEGIN
      WHILE NOT EOLN(infile) DO
         BEGIN
         outfile@ := infile@;      {define outfile@}
         PUT(outfile);             {write outfile@}
         GET(infile)             {get next character}
         END;
      READLN(infile);            {advance to next line}
      WRITELN(outfile)           {advance to next line}
      END
END.
```

7. USING FILES IN STRUCTURED VARIABLES

This implementation of Pascal does not fully support the use of files which are components of structured variables. The following declarations are examples of the use of files in structured variables:

```
VAR   files   : ARRAY [1..5] OF TEXT;        {array of files}
      student : RECORD
                name    : ARRAY[1..20] OF CHAR;
                scores  : FILE OF INTEGER;  {file in record}
                END;
```

The above declarations are legal but the I/O routines (see Chapter Ten of the Language Reference Guide) will not accept file names which are not simple. An example of a simple name is "**outfile**". With the above declarations, the file names are not simple names. I/O statements like the following would generate compile errors:

```
READLN(files[2],...
WRITELN(files[5],...
READ(student.scores,...
WHILE NOT EOF(student.scores) DO ...
```

For applications that need to use files as components of structures, there is a method of avoiding the simple name restriction to file names. Simply write your own I/O routines which act as interface to the Pascal I/O routines. You may pass non-simple file names to these interface routines which then use simple names in the actual I/O operations (see the example program on the following page). It is important to note that the file variables should be passed by reference (that is, they should be preceded by **VAR** in the parameter list).

When using files which are components of structures, make sure that the following two operations are always performed on the files:

1) Before the file is opened, it must be initialized. The following operation will initialize a file. The filename may be simple or non-simple.

   ```
   filename::INTEGER :=0          {initializes file "filename"}
   ```

2) Before the program is exited, the file must be explicitly closed by the **CLOSE** procedure. Failure to do so will probably result in loss of the file.

```
(*$NO INOUT*)
PROGRAM files_in_structures;

(* Sample program which uses array of files *)
(* This program prompts for a file name and
   then sends the file to the line printer *)

VAR   ch   : CHAR;
      files: ARRAY[1..2] OF TEXT;

PROCEDURE openr(VAR filename : TEXT);
BEGIN
   filename::INTEGER:=0;  {initialize file}
   RESET(filename)        {open file for reading}
END;

PROCEDURE openw(VAR filename : TEXT; name : STRING);
   PROCEDURE SETACNM(VAR f : TEXT; name : STRING); EXTERNAL;
BEGIN
   filename::INTEGER:=0;  {initialize file}
   SETACNM(filename, name);      {eliminate prompt for filename}
   REWRITE(filename)      {open file for writing}
END;

PROCEDURE closefile(VAR f : TEXT);
BEGIN
   CLOSE(f)       {close file}
END;

PROCEDURE readfile(VAR f : TEXT; VAR data : CHAR);
BEGIN
   READ(f,data)   {read from file}
END;

PROCEDURE writefile(VAR f : TEXT; data : CHAR);
BEGIN
   WRITE(f,data)  {write to file}
END;

PROCEDURE writeline(VAR f : TEXT);
BEGIN
   WRITELN(f)     {advance to next line of output file}
END;

PROCEDURE readline(VAR f : TEXT);
BEGIN
   READLN(f)      {advance to next line of input file}
END;

FUNCTION endfile(VAR f : TEXT) : BOOLEAN;
BEGIN
   IF EOF(f) THEN endfile := TRUE else endfile := FALSE
END;
```

```
FUNCTION endline(VAR f : TEXT) : BOOLEAN;
BEGIN
   IF EOLN(f) THEN endline := TRUE else endline := FALSE
END;

BEGIN
   openr(files[1]);                  {open input file}
   {note to CP/M users --> :L should be changed to LST:}
   openw(files[2],BLDSTR(':L'));       {open output file}
   WHILE NOT endfile(files[1]) DO
      BEGIN
      WHILE NOT endline(files[1]) DO
           BEGIN
           readfile(files[1],ch);
           writefile(files[2],ch);
           END;
      readline(files[1]);
      writeline(files[2])
      END
END.
```

8. USING GLOBAL VARIABLES IN EXTERNAL ROUTINES

It is recommended that whenever possible, variables should be passed to routines rather than allowing the routines to access global variables. However, sometimes the use of global variables is necessary. When using global variables in an external routine (that is, a routine that was compiled separately), it is necessary to duplicate the exact global environment when the external routine is compiled. Otherwise, referencing of global variables within the external routine will not be correct.

The use of the compiler **INCLUDE** option is very helpful to insure that the declarations used in the main program are exactly duplicated in the separately compiled routine. The following example illustrates the use of global variables in a separately compiled procedure.

File containing the main program:

```
PROGRAM main;
{the file GLOBAL contains the declarations for the main program}
{$INCLUDE 'GLOBAL'}
PROCEDURE separate; EXTERNAL;
BEGIN
   letter:='a';
   digit:=10;
   separate;
END.
```

File containing the external procedure:

```
PROGRAM compile_separately;
{duplicate the global environment}
{$INCLUDE 'GLOBAL'}
PROCEDURE separate;
BEGIN
   WRITELN('letter = ',letter);
   WRITELN('digit = ',digit:2);
END;
BEGIN
   {$NULLBODY}
END.
```

The file **GLOBAL**:

```
VAR   letter : CHAR;
      digit  : INTEGER;
```

9. USING COMMON VARIABLES

Often when creating libraries, such as a set of graphics routines, it is difficult to avoid the need for using global variables. There is usually a routine which does some initial processing to define variables which are needed by many of the other routines in the library. If these variables are local to the routine, they become undefined when the routine terminates. Of course, these variables could be retained if they were included as parameters to the routine. However, often these variables are not pertinent to the functionality from an end user's point of view. Making them a part of the parameter list complicates the use of the library routines. Another alternative is to make these variables global. This is a problem too, because each programmer which uses the library must know of these variables and make appropriate declarations for them. Common variables offer a clean solution to this type of programming problem. They essentially provide the ability to use global variables in libraries without the need for programs which use the library to even be aware of their existence. The following example illustrates the use of common variables.

File containing library of routines:

```
PROGRAM library;
COMMON xscale, yscale : REAL;

PROCEDURE axis(xmin,xmax,ymin,ymax : REAL);
ACCESS xscale, yscale;
BEGIN
   xscale := 512/(xmax-xmin);
   yscale := 256/(ymax-ymin);
END;

PROCEDURE scale(x,y : REAL);
ACCESS xscale, yscale;
VAR  ix,iy : INTEGER;
BEGIN
   ix := ROUND(xscale*x);
   iy := ROUND(yscale*y);
   WRITE('original values: x,y = ',x:6:1,',',y:6:1);
   WRITELN('scaled values: x,y + ',ix:3,','iy:3);
END;

BEGIN
   {$NULLBODY)
END.
```

User program:

```
PROGRAM user;
VAR i : INTEGER;
PROCEDURE axis(xmin,xmax,ymin,ymax : REAL); EXTERNAL;
PROCEDURE scale(x,y : REAL); EXTERNAL;
BEGIN
   axis(0.0,10.0,0.0,5.0);
   FOR i := 0 TO 10 DO scale(i,i/2);
END.
```

SECTION E: NETWORK PASCAL SYSTEM REFERENCE GUIDE

TABLE OF CONTENTS

NETWORK PASCAL SYSTEM REFERENCE GUIDE

Before you study this section, you should have worked through the NETWORK PASCAL BEGINNER'S GUIDE (section A) and should be comfortable with the steps presented in that section. This REFERENCE GUIDE summarizes the options available under Network Pascal and provides information that will be useful to intermediate or advanced programmers.

1. DISKETTE INFORMATION

Network Pascal was designed specifically for use with TRS-80 Network Operating Systems. It includes a full-screen text editor, a compiler for the full Pascal language, and utility routines to move files to and from the disk drives located on the network Host. The network is not used during editing or compiling.

The Network Pascal disk contains the following programs:

NETPCL/CMD	The main editor/compiler program for Network Pascal
NETPCL/OV1 **NETPCL/OV2** **NETPCL/OV3**	Overlays for the Network Pascal editor/compiler. **NETPCL** loads these into memory as needed.
LINKLOAD/CMD	The Network Pascal linking loader. **LINKLOAD** is used to organize one or more object code files into an executable command module, and to run the module.
EXAMPLE/PCL	Source code for a short Pascal program, included on disk as an example.
TRSLIB/OBJ	An object file containing code for the TRS-80 Pascal Procedure and Function Library, as listed in Appendix II of this NETWORK PASCAL SYSTEM REFERENCE GUIDE.
STRINGS/OBJ	An object file containing code for the String Function Library, as listed in Appendix III.
RANDOM/OBJ	An object file containing code for the Random File Routines, as listed in Appendix IV.
RUN/CMD	A program used to execute Pascal object code. **RUN** includes object code for the Procedure and Function Library, the String Function Library, and the Random File Routines.

2. NETPCL: THE NETWORK PASCAL EDITOR/COMPILER

2.a. Editor Information

See the EDITOR GUIDE (section C of this documentation) for a complete description of the Network Pascal editor.

2.b. Compiler Overview

The Network Pascal compiler accepts the same Pascal source language as the Model III Pascal compiler. See the TUTORIAL (section B) and the LANGUAGE REFERENCE GUIDE (section D) for descriptions of the Pascal language. In the interest of saving space, one extension has been left out. Network Pascal will not accept **COMMON** declarations.

The compiler allows you to compile with or without generating object code. Sometimes it is desirable to compile without generating object code. The compiler will execute faster if no object code is generated, and so compiling without object code allows you to make a fast check for errors. Further, if the program and its object are together too large to fit in the memory of the student station, you can still use Network Pascal to edit the program and to check it for syntax errors and then use the Model III Pascal compiler at a stand-alone system to generate object code.

During compilation, the listing of the program is displayed on the screen or printer and any errors detected by the compiler are identified by code number. The meanings of these error codes are listed on pages D-88 and D-89 in the LANGUAGE REFERENCE GUIDE (section D of this Network Pascal documentation).

If no errors are found in the program, the message "**NO ERRORS DETECTED**" is displayed at the bottom of the listing. If errors are detected, then a message indicating the number of errors detected appears at the bottom of the listing. To return to the menu, press any key.

2.c. Memory Usage by the NETPCL Editor/Compiler

Each student station in the network should have 48K (49152 bytes) of memory, or more. Part of this memory is used for the Network Operating System. The remainder is used by Network Pascal. The memory used by Network Pascal is divided into 5 different parts.

1. A major part of the memory contains the code for the **NETPCL** program.

2. A second section is used for portions of the code that do not reside permanently in the student station. These sections of code are called <u>overlays</u> and are loaded only when they are needed. The overlays are much shorter than Network Pascal, so loading them takes much less time than loading the entire program. In Network Pascal, you can edit, compile, and transfer files with minimum impact on the Network.

3. The third part of memory contains the two memory files. This section is partitioned to contain both a source file and an object file. The boundary between these two files is not fixed, so either can grow to fill as much space as required, limited only by available memory. The memory files provide work space for the editor and compiler.

   The source file in memory is used by the editor. The text that you see on your screen when editing is the content of the source file. The source file is also used to provide input to the Network Pascal compiler. The source file is stored in a blank-compressed format to save space. The source file can be copied to the host disk to allow for future editing.

   The object file contains the object code generated by the Pascal compiler. This object code can be transferred to a disk file for execution.

4. The fourth section of memory contains the stack. This stack is used as work space for the editor and compiler. Under normal conditions, there will be plenty of space in the stack. The only time that you will run out of space in the stack is when compiling very complex programs. If this occurs, the message **"OUT OF STACK"** will be displayed. Press any key to return to the main menu, then get back into the editor program and alter the program to use less stack during compilation by reducing the complexity. Especially reduce the number of nested **CASE** statements and the number of levels of parentheses in expressions.

5. The fifth section of memory is the heap. The heap is used to contain the symbol table created by the compiler. These symbols represent the names of the variables, types, procedures, and other programmer-defined objects. There is enough heap space to contain approximately 100 symbols. This should be enough for most moderate-sized programs.

3. USING RUN TO EXECUTE A PROGRAM

After the Pascal program has been compiled, it may be executed by the **RUN** program. This program will directly execute the compiled object code without the use of the **LINKLOAD** program. To use **RUN**, you must first use the editor/compiler **M** command to move your object file to the host disk. Then use the **N** command to exit to the **"Network"** prompt.

The **RUN** command is entered at the **"Network"** prompt as:

RUN filename

where **filename** is the name of the object program as stored on disk. An extension of **/OBJ** is assumed by the **RUN** program. A drive number may be specified.

The **RUN** command includes the object code for the TRS-80 support routines (**SETPOINT**, **CLEARSCREEN**, etc.) This means that any of these routines can be called from the program executing.

The **RUN** command allows the amount of stack space to be specified on the command line. The stack is used by the Pascal program to store local variables and to save return addresses for procedure and function calls. This stack is allocated when the program is run, and the required size is determined by the number and type of variables declared, and by the number and sequence of procedure calls. Methods of estimating the amount of stack required are described later in this section.

Stack size may be selected by following the program name with the stack size, separated by a blank or comma. For example, the following line would cause the program **DATABASE** to execute with 15K (15360 bytes) of stack space. (No angle brackets around the number of K is required in the **RUN** command)

RUN DATABASE 15K

The stack size can be specified as a decimal or hexadecimal number. Hexadecimal numbers must have "#" as the first character. This is the same notation as is used in the Pascal language. The letter 'K' means 1024, so 8K is equivalent to 8 * 1024 or 8192. If no stack size is specified, then 1/2 of the unused memory is allocated for the stack and the other half for the heap. (The heap is the area of memory used by the Pascal program for dynamic memory storage as required by the procedures **NEW** and **DISPOSE**).

When execution of a program completes, the amount of stack and heap used is displayed on the screen. These numbers reflect the actual quantity of memory used during execution.

The first thing that a Pascal program normally does is to open the files **INPUT** and **OUTPUT**. When this happens, the prompts :

```
INPUT  =
OUTPUT =
```

appear on the screen. You may then enter the name of the file or device to be used when the program reads from **INPUT** or writes to **OUTPUT**.

If you simply press the **ENTER** key, then input and output devices will be the keyboard and the screen. When any file is opened by a Pascal program (by calls to **RESET** or **REWRITE**), a prompt will appear on the screen. To the left of the equals sign will be the name of the file being opened. You should type the name of the disk file or device to be associated with that file.

NOTE: The **INPUT** and **OUTPUT** prompts that accompany the execution of every program may be eliminated for a particular program by the use of the **$NO INOUT** option, described on page D-82 (in the LANGUAGE REFERENCE GUIDE).

The file names that you use to direct Pascal input and output are in the same format as normal filenames of the operating system in use. The disk drive specification is optional. Input and output to any Pascal file can be directed using physical devices as well as disk files. The name of the line printer is **:L** and the name of the video screen is **:C**. There is also a dummy device, such that if a file is associated with **:D**, then no actual output occurs. This is useful if you wish to run the program and discard some of its output.

> NOTE: **RUN/CMD** is also provided on Model III Pascal Disk One. You might find it advantageous to use **RUN** on stand-alone systems and **LINKLOAD** (as described in the BEGINNER'S GUIDE and below) on Network for the following reason: **RUN** requires that both **RUN** and the Pascal object code be loaded into the student station each time a program is executed. With **LINKLOAD**, only the Pascal object code needs to be reloaded each time. **LINKLOAD** remains loaded in the student station until you ask to return to the **"Network"** prompt. Therefore, **LINKLOAD** cuts down on host access time.

4. USING THE PASCAL LINKING LOADER (LINKLOAD)

The Network Pascal linking loader (**LINKLOAD**) provides powerful ways to configure Pascal programs. Separately compiled programs and procedures may be loaded into memory together and executed. The program that results from linking can be stored as a command (**/CMD**) file on disk, allowing you to execute it directly from the **"Network"** prompt by simply typing the filename and pressing **ENTER**.

Unlike **RUN**, **LINKLOAD** does not contain object code for the Procedure and Function Library, the String Functions, or the Random File Functions. However, you can use **LINKLOAD** to load any or all of these files (**TRSLIB/OBJ**, **STRINGS/OBJ**, and **RANDOM/OBJ**) into memory with programs that need them, and to build them into the executable command file along with your program.

To invoke the linking load, type **LINKLOAD** when the **"Network"** prompt is displayed. At this point, the linking loader is brought into memory from disk. The first item displayed is a menu of commands followed by the command prompt (">>"):

```
L=Load, R=Run, N=Network, I=Init, S=Symbols, B=Build CMD
>>
```

To use one of these commands, type the single letter (or, optionally, the entire name) of the command and press **ENTER**. In cases where more information is required, prompts will appear. To redisplay the list of commands, enter "**H**" or "**?**".

4.a. The LOAD Command

The Load command is used to load programs, procedures, and functions into memory. One or more programs or sub-programs can be in memory at a time; this is what allows linking of programs.

To load a program, type **L** and press the **ENTER** key. The Load command will ask for a filename. Enter the name of the file. The file that you are attempting to load should contain object code as generated by the Pascal compiler. The object file will be opened, and the object code will be loaded into memory. Each time a program, procedure, or function is loaded, its name will appear on the screen. This allows you to monitor the load process, and shows the identity of the procedures being loaded.

LINKLOAD allows procedures that were compiled separately (see pages D-64 and D-65 of the LANGUAGE REFERENCE GUIDE) to be joined. A program may be compiled a piece at a time, and when changes are made only the parts affected by the change need to be recompiled. This allows the programmer to create libraries of utilities. These utilities can be loaded with any program that needs them, but need to be compiled only once.

4.b. The SYMBOLS Command

The Symbols command displays all currently defined or referenced symbols on the screen. One procedure name is displayed per line. After the procedure name is a character that describes the use of that procedure. A **"D"** indicates that the name is <u>defined</u>; that is, that the procedure has been loaded into memory. An **"R"** indicates that the procedure has been referenced but not yet defined. This means that a procedure that has been loaded makes a call to this procedure. All procedures that are called must be loaded before the program can run.

The last item on the line is the address of the symbol. If the symbol is defined (**"D"**), then this is the address in memory where the procedure begins. If the symbol has not been defined (**"R"**), then this is the address of the last place it was used (called).

4.c. The RUN Command

After a program has been loaded, it can be executed with the linking loader's Run command. As with the **RUN** program, one half of available memory is allocated to stack and the other half to heap by default. If these space allocations are sufficient, simply press **ENTER** when you see the prompt **STACK SIZE:**. Otherwise, enter a value. The size of the stack may be expressed as a decimal number of bytes, a hexadecimal number of bytes (precede the number with the **#** sign), or a number of kilobytes (example: **8K**). A kilobyte is 1024 bytes. Methods of estimating the required stack size appear on pages E-10 and E-11 in this section.

After the **STACK SIZE:**, **INPUT:**, and **OUTPUT:** prompts have been answered, the program will execute. If a file is opened by **REWRITE** or **RESET**, the Pascal filename will be displayed on the screen and you will need to enter the name of the actual file or device to be used. (This name can be the filename of a disk file, or else **:L** for the line printer, or **:C** for the screen and keyboard, or **:D** for a "dummy" device.)

> NOTE: Running the program using **LINKLOAD** is a good way to test program execution, but be aware that running the program clears the program out of memory. If you use Run, you'll have to reload all program segments before you use Build.

4.d. The BUILD Command

Once all of the parts of a program have been loaded, the master program may be saved on disk as a command file. (If you have used the Run command since the program's parts were loaded, you must repeat the loading process first.)

The Build (**B**) command is used to save the master program. The first prompt from this command is the same as for the Run command (**STACK SIZE:**) and has the same meaning. The Build command then asks for a filename. This is the name of the file that will contain the master program. The filename that you enter <u>must</u> contain the **/CMD** extension, and may contain a drive specification. For example:

MAINPROG/CMD:2

Once you have entered the filename, the program will be saved on disk, and you may run it at a later time simply by entering its name at the **Network** prompt. For example: **MAINPROG <ENTER>.**

The Build command then returns control to the Network Operating System.

4.e. The INIT Command

The Init command clears the symbol table and redisplays the command menu. This command can be used to start over if the wrong program is loaded by mistake. It is equivalent to exiting to the Network Operating System and then running **LINKLOAD** again.

4.f. The NETWORK Command

The **N** command returns control to the Network Operating System. The "**Network**" prompt will reappear.

4.g. LINKLOAD Error Messages

The following error messages can be generated by the linking loader.

***** CANNOT OPEN FILE**	This message is generated when you attempt to load and the loader cannot find a file of the name specified. This may be caused by a misspelling or by the wrong disk being in the drive.
***** UNRESOLVED REFERENCES**	When you use the Run command to execute a program or the Build command to generate an image on disk, the loader checks that all of the procedures that are called within the program have been loaded. If there are procedures or functions that have been called but have not been loaded, then this message is generated. At this point, you can load the required modules and repeat the command. The Symbols command can be used to list names of the procedures that are not yet defined. These will have an "R" in the listing.
***** INVALID OBJECT TAG**	This message is issued when a load is performed on a file that is not in a valid object format. The most frequent cause of this error is an attempt to load the source program instead of the object.
***** SYMBOL TABLE FULL**	The linking loader has room for 256 different external symbols. If more procedures than this are loaded, the symbol table will become full.
***** ILLEGAL REFERENCE**	This message signifies an inconsistent structure in an object file. It is an indication that the file has been damaged. The best solution is to recompile the offending program.

4.h. LINKLOAD Example

The listing on the next page illustrates a sample **LINKLOAD** session, as prompts from the computer and responses from the user might appear on the screen:

Network 3
LINKLOAD

```
L=Load, R=Run, N=Network, I=Init, S=Symbols, B=Build CMD
>> L
FILE     = MAIN/OBJ
MAIN
31557 BYTES LEFT
>> L
FILE     = TRSLIB/OBJ
GOTOXY
GETKEY
INKEY
CLEARSCR
CLEARGRA
WRITECH
WRITESTR
INP
GET$PROC
IO$ERROR
HP$ERROR
TIME
DATE
ITIME
SETPOINT
RSETPOIN
TESTPOIN
USER
CALL$
$MEMORY
NOBLANK
READCURS
PEEK
POKE
INIT$FIL
FILE$STA
SET$ACNM
30333 BYTES LEFT
>> B
STACK SIZE :
FILE     = MAINPROG/CMD
```

5. MEMORY USAGE FOR PROGRAM EXECUTION; ESTIMATING STACK SIZE

The Pascal **RUN** program or the linking loader is loaded beginning at address #5200 in the computer's memory. The Pascal program that is being executed will be loaded immediately above the runtime program.

The next segment above the program is used to contain the Pascal stack. Pascal programs use a stack to store local variables and to save return addresses for procedure and function calls. This stack is allocated when the program is run, and the required size is determined by the number and type of variables declared and the number of and sequence of procedure calls. The stack is a dynamic structure. Space is allocated when a procedure is called, and is released when the procedure is exited.

The remainder of available memory is used for the heap. Programs that use pointers and the procedure **NEW** will use storage from the heap. The heap also contains the buffers used to read from and write to files.

The total stack size required by a program is determined from its dynamic behavior at run time. Each time a procedure is called, space is allocated for its local variables. The total stack in use is a function of the number of procedures active at the time and the number and sizes of the variables used within those procedures. If two procedures are never active at the same time, then the space used by each can be shared. The total stack that must be allocated is determined from the maximum size that is in use at any given time.

The simplest way to determine stack requirements is to run the program. Specify enough stack for it to run, perhaps with an excess. When the program terminates, the maximum stack used by the computer is printed on the screen. A good rule of thumb is to allocate 20% more stack than is required for a typical execution of the program.

The size of stack required can also be determined from the source program. It is necessary to determine which procedures will be active at a given time. Then add the size of the local variables for each procedure. If too much or two little stack is allocated for the program, it may terminate unpredictably.

The sizes of simple variables are summarized below:

type	size in bytes
CHAR	1
BOOLEAN	1
INTEGER	2
STRING	2
REAL	4
REAL (double precision)	8
FILE	32
TEXT	32

The size of an array is determined by multiplying the number of elements in the array (upper bound minus lower bound, plus 1) by the size of the element. The size of a record is determined by adding the sizes of its individual fields. Packing is on byte boundaries.

A set may require from 1 to 33 bytes depending on the number of possible members. The set requires one byte plus one additional byte for each eight possible members. For example, a set with 16 possible members requires 3 bytes.

Enumerated types require one byte, and subranges require one or two bytes. (Ø..255 requires one byte.)

To calculate the total stack required, you should also include 64 bytes for the predeclared files **INPUT** and **OUTPUT**. Also, active procedures require space for their parameters as well as their local variables. Parameters passed by value require storage based on the size of the variable. Parameters passed by reference require two bytes each. Each active procedure also requires 9 bytes to store dynamic return information.

APPENDIX I: NETWORK PASCAL ERROR CODES

Network Pascal can detect several types of errors. These errors and their possible solutions are described below.

Editor Errors

UNKNOWN COMMAND	A command was entered in command mode that is not known to the editor. Check the spelling. Note that command names can be abbreviated, but that all characters that are actually entered must match with the name of the command.
MEMORY EXHAUSTED	The editor has run out of memory for the source file. Delete the object file, or remove some text from the source. You can also break the program into separate pieces that can be separately compiled, and then link them before execution.
STRING NOT FOUND	A "find string" command was executed and the string was not found in the section of the file between the cursor and the end of file. Either the string was mistyped, or it exists in the file before the cursor position, or the string is not present. Note that the editor begins with the string for the find string command set to null.

Compiler Errors

^ddd	(Where **d** represents a digit.) A one to 3 digit error code appears in the listing. Find the meaning of the error in the LANGUAGE REFERENCE GUIDE (pages D-88 and D-89) and correct it in the program.
OUT OF STACK	The compiler has run out of working storage space. Stack is used to evaluate expressions and keep track of nested statements. The solution is to reduce the amount of nesting in the program. The best methods are to reduce the number of levels of parentheses in expressions, or reduce the number of nested **CASE** statements.
OUT OF HEAP	The compiler has run out of space to store symbols. Either reduce the number of names present, or split the program so that portions of it can be compiled separately.

RUNTIME ERROR 09 The compiler has run out of memory to store the object code. Check to be sure that there are not two programs concatenated in the source file. If the source and the object together are too big to fit in memory, then compile the source without generating object code, to detect errors. When all errors are corrected, use the Model III Pascal compiler to compile and generate object code.

Move errors

Can't open file Move was unable to open the file. If a read was being done, it means that the file is not present on the disks located on the host. Check the spelling of the filename and the identity of the disks located on the host. If a write was being done, then it is not possible to create the file. The host disk may be write-protected, or out of directory space.

OUT OF SPACE The memory in the student station is full. Check that you are not inadvertantly concatenating files (that is, failing to delete an existing file before moving a new one into memory). Also check for a leftover file such as an object file from a previous compile or from a previous user of that station.

Move error An operating system error was detected during the move. This is usually a disk error (for example, disk full). Correct the error and try again.

1. For variable types to match in expressions and not generate compiler error messages, they must be explicitly declared to be of the same type in the variable declaration section. For example:

```
PROGRAM TEST;
VAR A:ARRAY[1..5]OF CHAR;
    B:ARRAY[1..5]OF CHAR;
BEGIN
    A := B;
END.
```

   will generate a type conflict message by the compiler although the types appear to match. The following example will not generate an error message and is perfectly legal in Pascal.

```
PROGRAM TEST;
VAR A,B:ARRAY[1..5]OF CHAR;
BEGIN
    A := B;
END.
```

   or:

```
PROGRAM TEST;
TYPE D = ARRAY[1..5]OF CHAR;
VAR A,B : D;
BEGIN
    A := B;
END.
```

2. A procedure declaration that has non-named types in the parameter list is illegal in Pascal. (That is, the following is illegal.)

```
PROGRAM TEST;
    PROCEDURE EXAMPLE(VAR A:ARRAY[1..5]OF CHAR); EXTERNAL;
BEGIN
END.
```

   The following is legal:

```
PROGRAM TEST;
TYPE D = ARRAY[1..5]OF CHAR;
    PROCEDURE EXAMPLE(VAR A:D); EXTERNAL;
BEGIN
END.
```

3. Placing a semicolon character (;) before the **ELSE** part of an **IF THEN ELSE** statement is illegal.

APPENDIX II:

TRS-80 PASCAL PROCEDURE AND FUNCTION LIBRARY

A set of functions and procedures to access the hardware features of the TRS-80 is provided with the Pascal compiler. These procedures can be declared as external procedures within Pascal programs. The object code for these procedures and functions is provided in two forms:

1. If the program is executed with the **RUN** command, the function library is contained within the **RUN** program. Any of the library procedures and functions can be called and the routine will be linked to when the program is loaded.

2. If the linking loader is used, these routines are not automatically available. This allows programs that do not need these routines to have more space available. The function library is provided in object form on disk. This file (**TRSLIB/OBJ**) can be loaded using the load command from the linking loader. This will make all of the library routines available.

Each of the library routines is described below. A Pascal external declaration is given. This declaration should be included in any program that uses the routine.

NOTE for Model III (non-network) Pascal Users Only: The external declarations of the library routines are included in a file on Model III Pascal Disk One (**TRSLIB/PCL**). Any or all of these declarations can be inserted into the source program using the insert file command available through the Model III Pascal **ED/CMD** program.

PROCEDURE CLEARGRAPHICS; EXTERNAL;

The purpose of this procedure is to clear the display when utilizing the graphics routines. Its function is similar to the **CLEARSCREEN** function (see page E-19) but it loads all hex 80's into the display memory, instead of hex 20's as **CLEARSCREEN** does.

PROCEDURE SETPOINT(X, Y : INTEGER); EXTERNAL;

This procedure sets a graphics point on the screen. The location of the point is specified with the **x** (horizontal) and **y** (vertical) coordinates. The value of **x** should be in the range **0 <= x <= 127**. The value of **y** should be in the range **0 <= y <= 47**.

PROCEDURE RSETPOINT(X, Y : INTEGER); EXTERNAL;

This procedure clears a graphics point on the screen. The location of the point is specified with the **x** (horizontal) and **y** (vertical) coordinates. The value of **x** should be in the range **0 <= x <= 127**. The value of **y** should be in the range **0 <= y <= 47**.

FUNCTION TESTPOINT(X,Y:INTEGER) : BOOLEAN; EXTERNAL;

This function tests the state of a point on the screen in graphics mode. **X** and **y** are the horizontal and vertical coordinates of the point to be tested. The function returns TRUE if the point is on (white), and FALSE if the point is off.

TYPE
  BYTE = 0..255;
  FUNCTION PEEK(ADDRESS : INTEGER) : BYTE; EXTERNAL;

This function returns the contents of any memory location. It may be used to examine memory or memory-mapped input devices. **ADDRESS** is the address being examined. An address may be passed if its value is known. The addresses of Pascal variables may be obtained by calling the **LOCATION** function, described on page D-68 in the LANGUAGE REFERENCE GUIDE.

PROCEDURE POKE(ADDRESS : INTEGER; VALUE : BYTE); EXTERNAL;

POKE is used to alter the contents of any location in memory. It may also be used to write to memory-mapped output devices such as the printer port.

PROCEDURE GOTOXY(X, Y : INTEGER); EXTERNAL;

GOTOXY positions the cursor on the screen at the specified location. If a write is performed to a file that represents the screen, the text will appear beginning at the addressed location. Procedures **WRITECH** and **WRITESTRING** described below also use this location. The value of **x** should be in the range **0 <=x <= 63**. The value of **y** should be in the range **0 <= y <= 15**. If **GOTOXY** is used with the **READ** or **WRITE** statements, then a call to the external procedure **NOBLANK** at the beginning of the program is necessary. The TRS-80 ROM driver for the screen will automatically clear the next line on the display when the carriage return character is received. This can be detrimental when constructing menu displays. A call to **NOBLANK** will cause the next line to always be redisplayed.

PROCEDURE NOBLANK(REDISPLAY : BOOLEAN); EXTERNAL;

The TRS-80 ROM routine driver for the screen will automatically clear the next line on the display when a carriage return character is received. A call to **NOBLANK** with **REDISPLAY := TRUE** will cause the next line to always be preserved. If **REDISPLAY := FALSE**, it will be blanked. The Pascal logical files used for screen display must be **RESET** after the **NOBLANK** call for it to take effect. This includes the default file **INPUT**.

PROCEDURE READCURSOR(VAR X, Y :INTEGER); EXTERNAL;

This procedure returns the current position of the cursor on the CRT screen. **X** is the horizontal position (character) and **y** is the vertical position (line).

PROCEDURE WRITECH(CH : CHAR); EXTERNAL;

This procedure writes a single character to the computer screen at the current cursor location. The cursor location is advanced by one.

```
TYPE
   CHARSTRING = PACKED ARRAY [1..XX] OF CHAR;
   PROCEDURE WRITESTRING(VAR S : CHARSTRING; FIRST, LAST : INTEGER);
      EXTERNAL;  (*XX IS ANY LENGTH*)
```

This procedure writes a portion of a string of characters to the screen. The text is written starting at the current cursor location. **FIRST** is the index of the first character to be written, **LAST** is the index of the last character to be written. The total number of characters displayed is: **LAST-FIRST+1**. If last is less than first, then no character is written.

PROCEDURE CLEARSCREEN; EXTERNAL;

A call to **CLEARSCREEN** causes the screen display to be cleared and the display to be set to 64 character width.

PROCEDURE INKEY(VAR CH : CHAR; VAR READY : BOOLEAN); EXTERNAL;

This procedure scans the keyboard to determine if a key is being pressed. If a key is currently pressed, then **CH** is the character generated by that key and **READY** is set to TRUE. If no key is pressed, then **READY** is FALSE, and **CH** is the space character (one blank space).

FUNCTION GETKEY : CHAR; EXTERNAL;

This function waits for and returns the next character from the keyboard.

FUNCTION INP(PORT : BYTE) : BYTE; EXTERNAL;

This function performs input from a Z80 I/O port. The port number is passed to the function and the value read from that port is returned as the function value.

PROCEDURE OUT(PORT, VALUE : BYTE); EXTERNAL;

This procedure performs physical output to a Z80 port. It may be used in conjunction with the function **INP** to communicate with devices interfaced as input or output ports. The two parameters specify the port number and the value to be written to that port.

FUNCTION FILE$STATUS(VAR F : TEXT) : BYTE; EXTERNAL;

This function returns the status of a file. The file can be of any type, but the external declaration must specify a type that matches the type of the file being tested. The byte returned is the error code for the latest I/O (input or output) error. If no errors have occurred, then zero is returned. This function is used in conjunction with **IO$ERROR** and allows a program to detect and recover from its own IO errors.

**PROCEDURE IO$ERROR(NEWSTATE : BOOLEAN;
VAR OLDSTATE : BOOLEAN); EXTERNAL;**

This procedure sets the state of the I/O error recovery flag within the Pascal runtime system. This flag is used to determine whether a program detects its own I/O errors. If the flag is set to TRUE, then default error processing is performed. In case of an error on a file or device, a message is displayed on the screen and the program halts.

If the I/O error flag is set to false, then all I/O errors are ignored by the system, and it is up to the program to check for and recover from I/O errors. I/O errors can be detected by calling the function **FILE$STATUS.**

NEWSTATE is a boolean value that sets the new state of the I/O error recovery flag. **OLDSTATE** is used to return the previous value of the flag. This allows a program to change the state temporarily and then restore it.

**PROCEDURE HP$ERROR(NEWSTATE : BOOLEAN;
VAR OLDSTATE : BOOLEAN); EXTERNAL;**

This procedure sets the state of the heap error recovery flag within the Pascal runtime system. When this flag is set to true, then a call to the procedure **NEW** will cause the program to terminate with an error message if no more space is available. Setting this flag to FALSE causes the procedure **NEW** to return **NIL** if no space is available. The calling program should check for **NIL** on each call to **NEW** when this flag is set to FALSE. This allows a program to use maximum memory from the heap without danger of an abnormal termination when space is exhausted.

NOTE: Both **NEWSTATE** and **OLDSTATE** must be initialized before **IO$ERROR** or **HP$ERROR** is called.

PROCEDURE $MEMORY(VAR STACK, HEAP : INTEGER); EXTERNAL;

This procedure allows a program to determine the amount of memory currently available. The parameter **STACK** returns the current number of stack bytes available and the procedure **HEAP** returns the amount of heap available.

```
TYPE
   FILENM = PACKED ARRAY[1..XX] OF CHAR;
   ALPHA  = PACKED ARRAY[1..8] OF CHAR;
(* XX IS ANY LENGTH LONG ENOUGH FOR THE FILENAME *)

PROCEDURE SET$ACNM (VAR F : TEXT; VAR filename :FILENM;
   NAMELENGTH : INTEGER; VAR FILEID : ALPHA) ; EXTERNAL;
```

SET$ACNM is used to set the name of the physical file or device to be associated with a Pascal file. It allows a program to compute file names internally. For example, a database program may know the name of the file containing the database. This procedure allows the program to specify the filename rather than requesting it from the keyboard.

The parameter **F** can be a file of any type. The external declaration of **SET$ACNM** that is included in the source program must specify a type for **F** that matches the actual file type to be used.

Filename is a string containing the text of the filename. This string must be compatible with the operating system syntax for filenames. The physical devices (lineprinter = **:L**, computer screen = **:C**, and dummy device = **:D**) may also be used. **NAMELENGTH** is an integer that specifies the length of the filename.

FILEID is an eight-character string that is used to identify the Pascal name for the file, such as **INPUT** or **OUTPUT**.

If **SET$ACNM** is called prior to a **RESET** or **REWRITE** on a file, then Pascal will not prompt for a filename. All subsequent uses of **RESET** or **REWRITE** will not cause a prompt unless a **CLOSE(filename)** is performed on the file. The filename association will remain as previously defined by **SET$ACNM**.

(Example program segment)

```
TYPE
FILENAME = PACKED ARRAY [1..15] OF CHAR;
ALPHA    = PACKED ARRAY [1..8]  OF CHAR;
VAR   FNAME  : FILENAME;
      FILEID : ALPHA;
      F      : TEXT;
PROCEDURE SET$ACNM(VAR F:TEXT; VAR FNAME:FILENAME; LEN:INTEGER;
                   VAR FILEID:ALPHA); EXTERNAL;
   BEGIN
      (* THIS ASSIGNMENT STATEMENT REQUIRES THE NAME TO BE LEFT *)
      (* JUSTIFIED, AND BLANK PADDED TO THE CORRECT ARRAY LENGTH*)
      FNAME := 'DATA/TXT:0     ';
      FILEID := 'F       ';
      SET$ACNM(F,FNAME,10,FILEID);
      RESET(F);
      READ(F,CH);
      (* AND ETC ........ *)
```

PROCEDURE SETACNM(VAR logical : **filetype;**
physical: STRING); EXTERNAL;

The library procedure **SETACNM** serves the same purpose as **SET$ACNM** but is simpler to use. The procedure takes only two parameters, the Pascal logical file variable, and the physical file or device name to associate with it. **Filetype** is any legal Pascal file type. The physical name parameter is a dynamic string. The **SETACNM** procedure disposes this string before exiting to recover the space.

If multiple file types are used in a program, the type transfer operator (::) may be use to allow **SETACNM** to be called with different file types. The external declaration of **SETACNM** may specify one of the file types used. The type transfer operator must then be used with the other file types to avoid a type mismatch error during the compile. Each of the other types must be type transferred to the same type as the one used in the declaration. The following example illustrates the use of **SETACNM.**

```
(*$NO INOUT*) (*eliminate the prompt for INPUT and OUTPUT*)
PROGRAM sample;

VAR printer   : TEXT;
    out       : FILE OF INTEGER;

PROCEDURE SETACNM(VAR F : TEXT; name : STRING); EXTERNAL;

BEGIN        (*main body of program sample*)
   (*map logical file "printer" to the line printer*)
   SETACNM(printer,BLDSTR(':L'));
   (*no prompt will occur when REWRITE(printer)is executed*)
   REWRITE(printer);
   (*map logical file "out" to disk file "OUT/DAT"*)
   SETACNM(out::TEXT,BLDSTR('OUT/DAT'));
   (*no prompt will occur when REWRITE(out) is executed*)
   REWRITE(out);
   ...
   ...
   END.  (*end of program sample*)
```

PROCEDURE USER(ADDRESS : INTEGER; VAR DATA : INTEGER); EXTERNAL;

This procedure interfaces to assembly language routines resident in the TRS-80's memory. **ADDRESS** is the physical address where the routine is loaded. Any assembly language routines that are to be called from Pascal should be loaded in a portion of memory that is not used by TRSDOS (or the Network Operating System) or Pascal.

On a stand-alone Model III or 4, the location of the top of memory can be set by using the TRSDOS utility **DEBUG**, to change the value in location #4411 on the Model III or 4. Pascal will not use any memory above this address, so assembly language routines can be loaded there. DEBUG cannot be used on a Student Station that is participating in a Network situation. The address of the highest usable location in memory for a student station of 48K or more is 65535 (decimal). Your assembly language routine should be loaded in as close as possible to the top of memory, to avoid conflicts with your own Pascal program and the runtime program you are using.

Information is passed to the assembly language routine through the **DATA** parameter. When the assembly language routine is called, the **HL** register pair contains the value of **DATA.** When the routine exits, the content of the **HL** register pair is returned as the new value of **DATA.** In cases where more than one word of information is required, the value of **DATA** can be the address of a variable. The address of any Pascal varible can be obtained by a call to the predefined function: **LOCATION.** This enables the called assembly language routine to access arrays or buffer data areas. The assembly language routine is entered with a standard Z80 call instruction and should be exited via a return. All Z80 registers are available for use in the assembly language subroutine.

PROCEDURE CALL$(ADDRESS : INTEGER; VAR A, STATUS : BYTE;
VAR BC, DE, HL, IX, IY : INTEGER); EXTERNAL;

This procedure can be used in a similar manner to **USER** to call assembly language subroutines. The difference is that **CALL$** allows you to set up all of the Z80 registers from Pascal. The values passed will be in the registers when the subroutine is called. When the subroutine returns, the current content of each register is returned to the Pascal program via the reference parameters. **STATUS** is the Z80 flag register.

TYPE
ALPHA = PACKED ARRAY [1..8] OF CHAR;
PROCEDURE TIME(VAR T : ALPHA); EXTERNAL;

This procedure returns the current time of day, as known to the Student Station. The time is in the form **HH:MM:SS.** (See the Network Operating System manual for information on setting the time at a Student Station.)

TYPE
ALPHA = PACKED ARRAY [1..8] OF CHAR;
PROCEDURE DATE(VAR T : ALPHA); EXTERNAL;

This procedure returns the current date, as known to the Student Station. The time is in the form **MM/DD/YY.** (See the Network Operating System manual for information on setting the date at a Student Station.)

APPENDIX III:

STRING FUNCTION LIBRARY

The following functions are provided for handling dynamic string manipulations. (See pages D-99 through D-101 of the LANGUAGE REFERENCE GUIDE for additional information.)

FUNCTION LEN(S : STRING) : INTEGER;

This function returns the length of a string.

FUNCTION LEFT$(S : STRING; POSITION : INTEGER) : STRING;

This function returns the left portion of the string, ending at the specified position within the string.

FUNCTION RIGHT$(S : STRING; POSITION : INTEGER) : STRING;

This function returns the right portion of the string, starting at the specified position and including the number of characters specified by length.

FUNCTION MID$(S : STRING; POSITION, LENGTH : INTEGER) : STRING;

This function returns the portion of the string starting at the specified position and including the number of characters specified by length.

FUNCTION STR$(LENGTH : INTEGER; CH : CHAR) : STRING;

This function returns a string of the specified length which is filled with the specified character.

FUNCTION ENCODEI(N : INTEGER) : STRING;

This function returns a string which is the character representation of the specified integer.

FUNCTION ENCODER(R : REAL) : STRING;

This function returns a string which is the character representation of the specified real. For single precision.

FUNCTION ENCODED(R : REAL) : STRING;

Same as **ENCODER**, but for double-precision reals.

FUNCTION DECODEI(S : STRING) : INTEGER;

This function returns an integer number which is the binary representation of the specified string.

FUNCTION DECODER(S : STRING) : REAL;

This function returns a real number which is the binary representation of the specified string. For single precision.

FUNCTION DECODED(S : STRING) : REAL;

Same as **DECODER**, but for double-precision reals.

FUNCTION CHARACTER(S : STRING; POSITION : INTEGER) : CHAR;

This function returns the character at the specified position in the string.

TYPE COMPAREVALUE = (LESS, EQUAL, GREATER);
FUNCTION CMPSTR(S1, S2 : STRING) : COMPAREVALUE;

This function compares the two specified strings and returns an enumerated value based on the comparison. The returned value is **LESS** if S1<**S2**, **EQUAL** if **S1=S2**, and **GREATER** if S1>**S2.**

FUNCTION CONC(S1, S2 : STRING) : STRING;

This function returns a string which is the result of the concatenation of the two specified strings.

FUNCTION CPYSTR(S : STRING) : STRING;

This function returns a copy of the specified string. The typical use for this function is the assignment of one string variable to another. This prevents both string variables from referencing the same string. For example, **STRING1 := CPYSTR(STRING2)**; will cause **STRING1** to refer to a different copy of **STRING2. STRING1 := STRING2**; causes **STRING1** to refer to the same copy of **STRING2** and any changes in the value of **STRING1** would cause **STRING2** to change also.

FUNCTION DELETE(S : STRING; POSITION, LENGTH : INTEGER) : STRING;

This function returns the string which results after deletion of a specified number of characters beginning at the specified position in the string.

FUNCTION FIND(SUBSTRING, S : STRING) : INTEGER;

This function returns an integer number which points to the start of the specified substring within the specified string. If the string does not contain the substring, then the returned value is 0.

FUNCTION INSERT(SUBSTRING, S : STRING; POSITION : INTEGER) : STRING;

This function returns a string which is the result of inserting the specified substring into the specified string at the specified position.

FUNCTION REPLACE(OLDSTRING, NEWSTRING, S : STRING) : STRING;

This function returns the string which results after replacing the old substring with a new substring within string S.

APPENDIX IV:

RANDOM ACCESS FILES

Random access refers to a file access method where any record may be read or written to in any order. Pascal does not define the Random file type. The following Pascal procedures and functions will allow random access to files on the TRS-80. The following Pascal routines are supplied with this package, in object code format on disk in the file **RANDOM/OBJ.** When using random access files, you should declare these routines as external in the main program. Then simply link to the supplied object file of random access routines (**RANDOM/OBJ** on the Network Pascal disk and on Model III Pascal Disk Two) with the linking loader to satisfy any external references.

The object code for random files is built into the **RUN** command.

The following declarations should be included in the source program.

A. RANDOM FILE ROUTINES

**PROCEDURE OPERAND(VAR F:FILETYPE; RECORDLEN:INTEGER; PATHNAME:STRING;
VAR STATUS:INTEGER); EXTERNAL;**

The purpose of this routine is to open a random file. The **F** variable is of any fixed type. Random file types are fixed in length and should be declared as a **FILE OF DATATYPE.** A text file is not a particularly useful **DATATYPE.** The filetype may be any structure such as an **ARRAY**, **RECORD**, etc. **RECORDLEN** must be the size required for the datatype. The **SIZE(J)** function may be used to determine the **RECORDLEN. PATHNAME** is the physical filename on disk. You must prompt the user if it is to be changed at runtime. **STATUS** is a code returned by Pascal or the operating system. The status code returns the status of an operation on a random file.

**PROCEDURE READRAND(VAR F: FILETYPE; RECORDNUM: INTEGER; VAR DAT:DATATYPE;
VAR STATUS:INTEGER); EXTERNAL;**

This routine is used to **READ** data from a random file. The **RECORDNUM** is the record number to be read. **DAT** is the buffer for the data and is declared to be of the same type as the components of the filetype (for example, if **FILETYPE = FILE OF INTEGER;** then **DATATYPE = INTEGER;**).

**PROCEDURE WRITERAND(VAR F:FILETYPE; RECORDNUM: INTEGER; VAR DAT: DATATYPE;
VAR STATUS: INTEGER); EXTERNAL;**

This routine is used to **WRITE** data to a random file. The **RECORDNUM** is the record number to be written. **DAT** is the buffer for the data and is declared to be of the same type as the components of the filetype.

PROCEDURE CLOSERAND(VAR F:FILETYPE); EXTERNAL;

Random files on TRSDOS are required to be closed before program termination. Failure to do so may result in a loss of data.

As with any operating system, there are some peculiarities about random files. For example:

1. If you **WRITE** record number 1 and **WRITE** record number 100, and then read any record from 2 to 99, the returned buffer will contain trash. The data will be whatever was previously on the diskette -- probably the contents of an old file. This is because the operating system does not keep that much context. It is up to the user to keep track of unwritten records so that they are not read.

2. Random file record sizes may be from 1 to 256 bytes only. All blocking is taken care of by the system.

3. The standard functions **EOLN**, **EOF** have no meaning for random files. The status codes as returned by the above routines perform those functions where applicable.

4. The procedure **OPENRAND** is used to open a file for reading and writing. Opening an empty file and reading is perfectly legal. It is up to the user to check the returned status on all random file operations.

5. Random file record numbers are defined from 0..32767.

6. As with normal files, if a file is declared locally within a procedure and opened (not passed in as a parameter), once the procedure is exited Pascal will automatically close the file using the standard **CLOSE** file routine for non-random files, and will position the **EOF** marker in the directory at the last record read or written. This may not be the correct position as desired by the program. An explicit call to **CLOSERAND** should always be used to close the random file and position the **EOF**. This will always correctly place the **EOF** mark.

7. You may declare a file to be:

```
(*where XX is any record length from 1 to 256*)
TYPE LINE = ARRAY[1..XX] OF CHAR;
VAR  F: FILE OF LINE;
```

Once the file has been opened, you may access it by using the **READRAND** and **WRITERAND** external procedures, even if the file was not created by Pascal. There is only one procedure for opening random files (no reset or rewrite). You may read or write to a random file.

B. RANDOM FILE ERROR CODES RETURNED BY EXTERNAL PROCEDURES

Generated by the operating system:

```
15   DISK WRITE PROTECTED
24   FILE NOT FOUND
27   DISK FULL
28   END OF FILE
29   RECORD NOT FOUND (PAST EOF)
```

Generated by the random file routines:

```
128  PATH NAME IS NULL OR TOO LONG
129  RECORD LENGTH IS NOT BETWEEN 1 AND 256
130  FILE IS ALREADY OPEN
131  FILE IS NOT OPEN
```

If multiple random file types are used in a program, the type transfer operator (::) may be used to allow the random file routines to be called with different file and data types. The declarations may specify one of the file and data types used in the program. Any other file and data types must be type transferred to the same types used in the declarations to avoid a type mismatch error during the compile. The following example illustrates the use of the random file routines. The status may be checked after each random file operation to determine if an error occurred. The returned status will be 0 if no error is detected during an operation.

```
PROGRAM sample;
TYPE file1 = FILE OF CHAR;
     file2 = FILE OF INTEGER;
VAR  f1      : file1;
     f2      : file2;
     value1, ch : CHAR;
     value2, status, number : INTEGER;
   PROCEDURE OPENRAND(VAR f : file1; length : INTEGER;
        name : STRING; VAR status : INTEGER); EXTERNAL;
   PROCEDURE CLOSERAND(VAR f : file1); EXTERNAL;
   PROCEDURE READRAND(VAR f : file1; number : INTEGER;
        VAR data : CHAR; VAR status : INTEGER); EXTERNAL;
   PROCEDURE WRITERAND(VAR f: file1; number : INTEGER;
        VAR data : CHAR; VAR status : INTEGER); EXTERNAL;
   PROCEDURE checktatus(status : INTEGER);
   BEGIN
     IF status <> 0 THEN
        WRITELN('* I/O ERROR: code number = ',status:3,' *')
   END;
BEGIN
   (* open file "F1/DAT" *)
   OPENRAND(f1,SIZE(CHAR),/BLDSTR('F1/DAT'),status);
   (* open file "F2/DAT" *)
   OPENRAND(f2::file1,SIZE(INTEGER),BLDSTR('F2/DAT'),status);
   FOR number := 0 TO 255 DO
```

(continued)

```
  BEGIN
  (* write the ASCII character set to F1/DAT *)
  ch := CHR(number);
  WRITERAND(f1,number,ch,status);
  (* write the ordinal values of the character set to F2/DAT*)
  WRITERAND(f2::file1,number,number::CHAR,status);
  END;
FOR number := 0 TO 255 DO
  BEGIN
  (* read the ASCII character set from F1/DAT *)
  READRAND(f1,number,value1,status);
  (* read the ordinal values of the character set from F2/DAT *)
  READRAND(f2::file1,number,value2::CHAR,status);
  END;
checkstatus(status);       (* check error status *)
CLOSERAND(f1);             (* close F1/DAT       *)
CLOSERAND(f2::file1);      (* close F2/DAT       *)
END.
```

SECTION F: MODEL III PASCAL SYSTEM REFERENCE GUIDE

TABLE OF CONTENTS

MODEL III PASCAL SYSTEM REFERENCE GUIDE

1. DISKETTE INFORMATION

Model III Pascal Disks One and Two are designed for use on a 48K TRS-80 Model III or Model 4 disk system. The TRSDOS 1.3 operating system is supplied on both of these diskettes. To execute under the LDOS operating system (floppy or hard disk versions), Model III Pascal must first be patched. See Appendix II of this guide for information on executing this patch. Model III Pascal may be used with the NEWDOS 2.0 and DOSPLUS 3.3 and 3.4 operating systems without patching.

For information on making a backup copy of each of the two stand-alone system diskettes, see the BEGINNER'S GUIDE. Never use the diskettes supplied with this package, except for making backup "working" copies.

Programs on the two Model III Pascal diskettes are:

<u>Model III Pascal Disk One</u>

PASCAL/CMD	The Model III Pascal compiler program.
RUN/CMD	The Model III Pascal "run" program (identical to the Network **RUN** program described in section E).
ERRORS/DAT	Contains text of error messages for Pascal compiler. You won't work directly with this file.
ED/CMD	The Model III Pascal editor program.
HELP/HLP **KEY/HLP** **CMD/HLP**	Contain helpful information that can be displayed to the user through the **ED/CMD** program, using the editor's **HELP** command. See page C-13 of the EDITOR GUIDE.
TRSLIB/PCL	Source code for external declarations of the Procedure and Function Library, as listed in Appendix II of the NETWORK PASCAL SYSTEM REFERENCE GUIDE.
STRINGS/PCL	Source code for external declarations of the String Function Library, as listed in Appendix III of the NETWORK PASCAL SYSTEM REFERENCE GUIDE.
PATCHER/CMD LDOS/PAT	Files used in completing the LDOS patch, as described in Appendix II of this section.
DATABASE	Source code for a database program, included on the diskette as an example.

T011/TMP	Under normal conditions, the user does not directly work with this file. It is used by the Model III Pascal editor program as a temporary storage file for the program being edited. In case an error occurs while you are exiting the editor program and the program you are editing becomes damaged, **T011/TMP** should still contain a good copy of the edited program.

Model III Pascal Disk Two

PASCALB/CMD	The overlaid compiler, designed to allow compilation of larger programs.
PASCAL/OV1 **PASCAL/OV2** **PASCAL/OV3** **PASCAL/OV4**	Overlays used by **PASCALB**. The overlaid compiler loads these in as needed.
LINKLOAD/CMD	The Model III Pascal version of the linking loader. Except for a few minor details covered in this section, Model III **LINKLOAD** works just like Network Pascal **LINKLOAD**, covered in section E.
CODEGEN/CMD	The Model III Pascal code generator -- an Advanced Development program covered in this section.
CODEINIT/CMD	A file used by **CODEGEN** for initialization. You will not need to work directly with this file, but it should be on the diskette when **CODEGEN** is used.
OPTIMIZE/CMD	The Model III Pascal optimizer -- an Advanced Development program covered in this section.
TRSLIB/OBJ	Object code for the Procedure and Function Library. This file can be linked to programs that use it, by the **LINKLOAD** program.
STRINGS/OBJ	Object code for the String Function Library. **LINKLOAD** can link this file to programs that use it.
RANDOM/OBJ	Object code for the Random File Routines, as listed in Appendix IV of the NETWORK PASCAL REFERENCE GUIDE. **LINKLOAD** can link this file to programs that use it.

2. DEMONSTRATION: USING MODEL III PASCAL ON A STAND-ALONE SYSTEM

Following along with this section will help you become familiar with using TRS-80 Pascal on a stand-alone Model III or Model 4 system. In this demonstration, we'll write a short source program, compile it into an object file that can be run, and then run the program.

Notice that this demonstration closely parallels the BEGINNER'S GUIDE for Network Pascal. This structure should help highlight for you the important differences between using Network Pascal and using TRS-80 Pascal for the stand-alone systems.

Steps in this demonstration may be completed using any Model III or Model 4 system with at least 48K of memory. Programs featured in this demonstration are on Model III Pascal Disk One.

> NOTE: If you are using the LDOS operating system, please turn to Appendix II of this guide for important steps you need to complete before using TRS-80 Pascal.

2.a. Using the Editor to Enter a Program

Let's begin by using the TRS-80 Pascal editor program (**ED/CMD**) to enter a Pascal program. Turn on the computer. Insert Model III Pascal Disk One into Drive Ø (the bottom drive) with the square notch to the left and the label facing up. Press the RESET button. When TRSDOS requests the date, enter the date in the form **MM/DD/YY**. When TRSDOS requests the time, simply press **ENTER**.

When **TRSDOS Ready** is displayed, type **ED** and press **ENTER**. In a moment the Pascal editor will be loaded. Since you did not specify a filename after "**ED**", the editor is ready for you to enter a new program, and a nearly blank screen is displayed:

***EOB**

"***EOB**" in the upper corner of the screen marks the "End of Buffer" (the bottom boundary of the temporary program storage area).

In order to enter text, we first need to insert at least one blank line. Blank lines may be added one at a time by holding down **SHIFT** and pressing **@**, or by pressing the **F1** key if you are using a Model 4. (Model III Pascal does not have the **CLEAR E** function to add 16 lines at once, though Network Pascal does.)

By moving the cursor to any position and then typing, we can insert text into the file, (providing that a blank line has been inserted at that position). If a character is overstruck, the old character is replaced by the new one. Now let's enter a program into memory by typing the program lines on screen, exactly as shown. At the end of each line, press **ENTER** to start a new line. To correct a typing error, backspace using the left-arrow key. For a complete list of the special keys that can be used in editing a program, see the Appendix to the EDITOR GUIDE.

```
PROGRAM test;
BEGIN
WRITELN('* I am a Pascal wizard.');
END.
```

Once you have correctly entered the program, press the **CLEAR** key followed by the letter **C**. A pair of angle brackets **< >** in the bottom left corner of the screen indicates that you are now in command mode, where you can type commands for the computer to follow immediately, without these commands being considered part of the program. Now type the command **exit** (or **EXIT**) and press **ENTER**. You'll see the message **<EXIT>FILE:**. Type the filename that you would like to save this program under, plus the extension **/PCL**, plus a drive number if desired. In the example below, we'll use the filename "**TEST**" and omit the drive number:

TEST/PCL

The editor will proceed to save the source program onto disk, and will then return to the operating system prompt.

NOTE: As a rule, you should always use the extension **/PCL** when saving your Pascal program. The simple form of the **PASCAL** command (used to compile a Pascal program) requires that the file to be compiled have this extension.

2.b. Using the Compiler to Compile the Program

Once the source code is saved on disk, the next step is to compile the program. The Pascal compiler translates the source program into a form that the computer can understand. At the same time, it checks for errors in the source program and reports any errors it finds to the programmer.

To compile our example program "**TEST**", type the following at the operating system prompt, and press **ENTER**:

PASCAL TEST

Note that the compiler assumes that the filename has the extension **/PCL**, and it appends this extension to the filename.

The command **PASCAL TEST** causes the operating system to load and execute the Pascal compiler; the compiler then translates the source code contained in the file **TEST/PCL** into object code that can be run on the computer. The compiler then stores the resulting object code in a file called **TEST/OBJ.** A listing is sent to the screen, showing the source program and error messages for any errors detected. If you made any typing errors in entering the program of this demonstration, you'll see a caret symbol (^) pointing to the line with the error, and an error code number will be given. At the end of the listing, the total number of errors is shown.

If your total number of errors is zero, you have successfully compiled the program and you can now execute the program. Skip to "RUNNING THE PROGRAM" at the bottom of this page.

If any errors are listed, you'll need to use the editor again to correct these errors, as described below. Then you'll need to compile the program again.

2.c. Using the Editor to Correct Programming Errors

Since our demonstration program is short, errors should be fairly easy to find. With the compiler listing still displayed, look closely at the lines where any errors are shown. If you see one error right away but do not see any later errors, don't be surprised. One error near the beginning of a program can cause the compiler to perceive errors in later lines that would otherwise be correct. (For example, spelling "**BEGIN**" as "**PEGIN**" causes a total of 3 errors.)

To correct errors, first get back into the editor program, making sure that you specify that you want to work with this program (type **ED TEST/PCL** and then press **ENTER**). Retype any incorrect characters, and then press **ENTER** at the end of each corrected line.

Exit the editor program (specifying **TEST/PCL** for the filename) and try compiling the program again. Repeat the process as many times as necessary until the program compiles without error.

Hint: When looking for errors, pay close attention to punctuation. Punctuation is very important in Pascal programs, and is easy for beginners to overlook.

2.d. Using RUN Program to Execute the Program

When the program is successfully compiled and the operating system prompt has reappeared, you can run the program using the program **RUN/CMD** on Disk Two of Network Pascal. The format used to run a program is **RUN filename** followed by a press of **ENTER**. For this demonstration, you'd enter **RUN TEST.** The **RUN** program assumes that the extension is **/OBJ. RUN TEST** causes the object code stored in the file **TEST/OBJ** to be loaded into memory and executed.

Next you'll see the message "**INPUT =**". Pascal is asking you to name your input file. **INPUT** and **OUTPUT** are discussed in the TUTORIAL, but for now just press **ENTER** for **INPUT** and **ENTER** for **OUTPUT**. This will send the output of the program (the message it prints) to the video display.

Once **INPUT** and **OUTPUT** are specified, the program runs and the **TRSDOS** prompt returns. You should see the message "*** I am a Pascal wizard.**" printed on the screen by the program.

You have now successfully entered, compiled, and run a short Pascal program. In addition, you have the source code for the program stored on disk under the filename **TEST/PCL** and the executable object code stored on disk under the filename **TEST/OBJ**. The permanent copy of the object code allows you to run the program in the future if desired. The permanent copy of the source code means that you can edit and recompile the program in the future if desired.

2.e. Summary

In brief, the process of developing a Pascal program on a stand-alone Model III or Model 4 is as follows:

1. Use the Pascal editor **ED/CMD** to develop source code.
2. Use the Pascal compiler **PASCAL/CMD** to generate object code and screen the program for errors.
3. If any errors are detected, use the Pascal editor to correct those errors; then use the compiler to recompile the program.
4. Optionally, use the **OPTIMIZE** or **CODEGEN** utilities to reduce the size of the program or to increase the speed of the program.
5. Execute the program by using the **RUN** module to execute the program, or by using the **LINKLOAD** utility to build and execute a command file.

Detailed information on the use of the TRS-80 Pascal editor can be found in Part Two of the EDITOR GUIDE. Detailed information on **RUN** and **LINKLOAD** can be found in the NETWORK PASCAL REFERENCE GUIDE. Details on the Model III Pascal compiler, and on **OPTIMIZE** and **CODEGEN**, can be found on the following pages.

3. THE MODEL III PASCAL COMPILER

The TRS-80 Pascal compiler is simply a program, written in Pascal, whose purpose is to translate the source code for other Pascal programs into an intermediate language called pcode. The pcode is a low-level language designed specifically to be the output of the Pascal compiler. Pcode resembles the assembly language for a stack-oriented computer. Once a program has been compiled, the pcode object program is stored in a file with the extension **/OBJ**. This object file may then be loaded and executed by the computer.

NOTE: When the compiler is running, you can abort compilation and return to TRSDOS by pressing the **BREAK** key.

3.a. The PASCAL Command

The **PASCAL** command causes the Pascal compiler to be loaded and executed. This command has two forms.

The short form:

The simplest form is:

PASCAL <stack> filename

where **filename** is the name of a file containing a Pascal program. The "**<stack>**" is an optional parameter that sets an upper limit on memory space that the compiler may use for stack manipulations. If stack is specified, angle brackets are required. If stack is not specified, a default stack size of 4K is used. 4K should be sufficient for most programs.

The extension for the source file is assumed to be **/PCL** and the object file is sent to a file of the same name but with the extension **/OBJ**. Any extension typed in the command line will be ignored. (And any source file whose filename does not include the extension **/PCL** will not be found by the compiler when the short form of the **PASCAL** command is used.) A disk drive may be specified.

As an example, the command:

PASCAL TEST:1

will cause the program **TEST/PCL** on drive 1 to be compiled and the object code to be stored on drive 1. When the disk drive is specified, the same disk drive will be used for both source and object. If the disk drive is not specified, the operating system will search all the drives for the first occurrence of **TEST/PCL**; the object code will be stored on the lowest-numbered drive that is not write-protected.

The short form of the **PASCAL** command always displays a listing on the computer screen.

The long form:

To use the long form of the **PASCAL** command, begin by simply entering the word **PASCAL** at the operating system prompt. To specify stack, you would enter **PASCAL <stack>**. For example:

PASCAL <3K>

Default for stack is 4K. The compiler then prompts for the names of the source and object code, and for the name of the disk file or device that is the destination of the listing. You should type the names of the files to be used. The source and object files can be on different drives. The listing can be placed in a disk file, sent to the screen (**:C**), or sent to a line printer (:L).

The compiler automatically outputs the lines of a program which contain errors to a file named **PASCAL/ERR.** If this file does not already exist, then the compiler creates it the first time an error is detected during a compile. For each line containing an error, the line number, the line itself, and the error number(s) are sent to this file. This makes it possible to discard the compiler listing (map to **:D**, the "dummy" device) and still be able to determine what errors were detected during a compile.

The following sequence will cause the file **TEST/TMP** to be compiled. The object code is then stored in **TEST/OBJ** on Drive 2, and the listing is sent to the line printer.

```
PASCAL
SOURCE  = TEST/TMP
LISTING = :L
OBJECT  = TEST/OBJ:2
```

3.b. The PASCAL Compiler Listing

The Pascal compiler reads the source program from a file and produces two outputs. One of these is a file containing the object code, used to execute the program. The other output of the compiler is the listing. The listing contains the text of the original source program, plus some additional information.

The listing is divided into "pages" with a heading at the top of each page. The heading contains the version number of the compiler and the page number. Each page after the first begins with a form feed [**CTRL L** or **0C** character], which will cause most printers to begin a new page. The number of lines per page is 62 unless this default page size is changed using the compiler option **PAGESIZE**, described on page D-85.

Each line of the listing is numbered, and the compiler also may generate hexadecimal addresses for each line of the program, if the user has invoked the compiler option **WIDELIST**, as described on page D-86. The hexadecimal addresses represent the locations of the generated object code, relative to the start of the program. If the program contains procedures or functions, the addresses for these routines are relative to the start of the routine.

If errors are detected by the Pascal compiler, error messages will appear in the compiler listing. An error message line, placed immediately below the line in error, begins with a string of five asterisks (*****) and includes a caret symbol (^) that points to the approximate location within the line where the error was detected. The caret symbol is followed by one or more error codes. It is possible for a single error to generate more than one error code, but in most cases the first error code identifies the cause of the error.

If any errors were detected, a summary of the meanings of the error codes that were generated appears at the end of the listing.

3.c. COMPILER MEMORY CONSTRAINTS AND THE OVERLAID COMPILER

The TRS-80 Pascal compiler requires approximately 33K of memory for code. Of this total, approximately 27K is the compiler itself and the remainder is runtime support. The runtime support portion contains the drivers for input and output devices, an interface to the file system, and the pcode interpreter. TRSDOS occupies 4.5K of memory, which leaves 10.5K bytes of memory for data in a 48K Model III microcomputer running TRSDOS.

4K of the remaining 10.5K is used for stack space by the compiler, with the result that the heap is about 6K -- enough space for about 250 symbols to be defined. A program that uses more than 250 symbols at a time will run out of heap space during the compile.

There are some ways of saving memory during the compile so that larger programs can be compiled. The limit on symbols is relative to the number of symbols visible at any point within the program. Symbols that are not available to the program are not retained by the compiler. The use of symbol table space can be improved by defining fewer global variables at the outer levels of the program, and making use of local variables whenever possible. This is also good programming practice.

The length of symbol names is not relevant to Pascal, unlike BASIC. Use of long names has no effect on program size or compiler memory usage. But extensive use of string constants will cause the compiler to use more memory. If a string constant is used in more than one place in a program, it will take less space if it is defined as a constant.

PASCALB is the overlaid, or segmented, version of the compiler. This version dynamically loads portions of the Pascal compiler from disk as needed. This increases the amount of memory available for symbols and allows longer programs to be compiled. The overlaid compiler will compile programs that are four times the size that can be compiled with the non-overlaid compiler (that is, a typical 4000 line program will compile successfully). The overlaid compiler does run more slowly than the non-overlaid compiler.

To invoke the overlaid compiler, use either format of the **PASCAL** command, substituting **PASCALB** for **PASCAL**.

4. USING THE PASCAL LINKING LOADER (LINKLOAD) ON A STAND-ALONE SYSTEM

The Model III version of **LINKLOAD** provided on Model III Pascal Disk Two is identical to the Network Pascal version of **LINKLOAD**, except as outlined below. Network **LINKLOAD** was described at length in section E (pages E-5 through E-9).

- The **"N=Network"** option listed in the Network **LINKLOAD** command menu appears in the Model III version as **"T=TRSDOS"**. Both return you to the operating system prompt.

- The Symbols command in Network **LINKLOAD** will label a "defined" procedure name with a **"D"** and a "referenced" procedure name with an **"R"**. Model III **LINKLOAD** has a third symbol -- **"C"** to indicate a common block. Commons are used to provide statically allocated shared data and are not available in Network Pascal. (Commons are discussed on pages D-16 and D-109 in the LANGUAGE REFERENCE GUIDE.)

5. ADVANCED DEVELOPMENT PROGRAMS -- THE OPTIMIZER

The optimizer is a program that takes the pcode generated by the compiler as input and outputs a compact, optimized form of the same pcode. Optimized pcode will execute faster than non-optimized pcode, but the main purpose of the optimization is to make the pcode more compact.

The amount of memory space gained by optimizing a program's pcode depends on the types of language features utilized by the original source program. Typically, the percent reduction in program size is between 10 and 30 percent. Quite often, this will enable the execution of a program that otherwise would abort due to lack of memory.

5.a. When to Use the Optimizer

The optimizer should be used any time program size is an important factor. The amount of memory required by a program is determined by the number of instructions to be executed and the number and sizes of the variables used. The factor that the optimizer addresses is the number and length of instructions. The greatest benefit will then be realized when optimizing long programs (>1000 lines).

In many cases, optimized code is slightly faster than non-optimized code, and even short programs will sometimes benefit enough to make optimization worthwhile. If a short program requires lots of data storage, optimization will maximize the amount of memory available for the data.

> NOTE: Only pcode object files may be optimized. Do not attempt to optimize command files (**/CMD**) or files generated by the code generator (**/COD**).

5.b. How to Use the Optimizer

Any pcode object file may be used as input to the optimizer program. That is, any compiled Pascal program or separately compiled portion of a Pascal program may be optimized. The optimizer assumes that the file to be optimized has the filename extension **/OBJ**, which is the extension that the stand-alone system version of Pascal automatically appends to each pcode object file at the time it is compiled.

The optimizer program is stored as a command file and therefore may be executed simply by typing **OPTIMIZE** from the top level of the operating system. Like the compiler, it can be invoked using either of two forms.

The Short Form:

OPTIMIZE filename

"**filename**" may include a drive specifier, but should not include an extension. The output of the optimizer (the optimized pcode) is placed in a file of the same name, but with the extension **/OPT**. If a drive was specified when the optimizer was invoked, the **/OPT** file will be placed on the same drive as the **/OBJ** file. Otherwise, the operating system decides which drive to use. The **/OPT** file may then be executed just like any **/OBJ** file is, using the **RUN** or **LINKLOAD** command.

The Long Form:

OPTIMIZE
LISTING = listingfile/ext or device (:C, :L, or :D)
INP_OBJ = inputfile/ext
OUT_OBJ = outputfile/ext

The long form requires that you enter the full filename, including extension, for both the input file (non-optimized) and the output file (optimized). The **LISTING** will show the name of each separate module in the input pcode file as it is processed. After each name will appear its original size in bytes followed by its optimized size in bytes. The **LISTING** may be directed to a file or a device (simply pressing **ENTER** will direct the **LISTING** to the computer screen). All filenames may include drive specifiers.

At completion, the optimizer program will display on the listing the size of the non-optimized pcode used as input and the optimized pcode generated as output.

ORIGINAL LENGTH = size in bytes
OPTIMIZED LENGTH = size in bytes

5.c. Example Use of the Optimizer

The following is an example of optimizing the **DATABASE** program supplied with the TRS-80 Pascal System (on Model III Pascal Disk Two). This example demonstrates use of the optimizer in both the short and long forms.

NOTE: Before working through this example and the later examples in this section, you should make a diskette that has plenty of free space for program storage. See Appendix III.

---

1. Compile the **DATABASE** program.

```
PASCAL   <ENTER>
SOURCE  = DATABASE  <ENTER>
LISTING = :C
OBJECT  = DATABASE/OBJ
```

2. Optimize the program's pcode in file **DATABASE/OBJ.**

<u>Short form example</u>

```
OPTIMIZE DATABASE
```

<u>Long form example</u>

```
OPTIMIZE
LISTING = :L
INP_OBJ  = DATABASE/OBJ
OUT_OPT  = DATABASE/OPT
```

Results:

Both the short form and the long form above would produce the same results. The pcode in file **DATABASE/OBJ** would be optimized and output to the file **DATABASE/OPT.** The short form would direct the listing to the computer screen, while the long form in this particular example would direct the listing to the line printer (because :L was used). The output would appear as shown below:

```
NOCUSTMR   15   13
PRESS      56   49
NEWSPACE   43   29
READDBAS  226  162
WRITEDBA  389  312
CUSTMROU  525  423
READTRAN  269  208
WRITETRA  415  325
DISPLAYD  114   84
LISTTRAN  130   91
LISTCUST   64   50
HEADING    70   64
MAINMENU  743  667
QUERYMEN  462  416
ADDCUSTM  193  153
```

QUERYTRA	**213**	**161**
ADDTRANS	**349**	**266**
SEARCHCU	**284**	**218**
QUERY	**170**	**132**
DATABASE	**302**	**264**

ORIGINAL LENGTH = 5437
OPTIMIZED LENGTH= 4418

6. ADVANCED DEVELOPMENT PROGRAMS -- THE CODE GENERATOR

The code generator is a program that translates pcode instructions to native machine code instructions. Any compiler-generated pcode object file or optimized pcode file may be used as input to the code generator program. Whole programs or separately compiled parts of programs may be "code-gened" to increase execution speed.

The speed increase realized by the code generator depends on the original program, but the typical code-gened program will be three to five times faster than the original program.

6.a. When to Use the Code Generator

The code generator increases execution speed by translating pcode instructions to machine instructions. Since each pcode instruction is equivalent to several machine language instructions, code generation also causes an increase in the size of the program. Therefore, the decision of whether or not to perform code generation on a program must not only be based on speed requirements, but also on program size. Typically, code generation will cause the size of the object to increase to two or three times the size of the original pure pcode program.

The execution speed of most programs will be adequate even when left in pcode form. However, programs which do lots of calculations within loops may benefit significantly from code generation. Also, when a program contains one or more procedures which are frequently called, code generation on these sections of the program can provide quite an improvement in execution speed.

To determine whether or not to code-gen a program, first run the program in pcode form. If the execution speed is determined to be slow, the next step is to determine whether to code-gen the whole program or selected parts of the program. As a general rule, small programs should be totally code-gened. The size increase for small programs will probably be insignificant. However, for large programs, the size increase may be very significant. For programs longer than about 1000 lines, an increase of two or three times in the size of the object code will significantly reduce the amount of memory left for the program data area (stack and heap). In cases where the size increase would not allow enough room for data area, selected procedures should be declared as externals and compiled separately.

The procedures selected for code generation should be the ones which most affect execution speed. After code generation, these procedures may then be linked to the main program using **LINKLOAD.** This process will allow for an increase in speed without causing the size to increase to a level that prevents the program from being executed.

The code generator performs most of the optimizations performed by the optimizer. Therefore, it is not necessary to optimize a program before performing code generation.

6.b. How to Use the Code Generator

Any compiler generated or optimized pcode file may be used as input to the code generator. The compiler generates files with the default extension of **/OBJ.** The optimizer generates files with a default extension of **/OPT.** Whole programs or separately compiled programs may be code-gened. In either case, simply compile the Pascal source and run the code generator program, using the compiler-generated object file as input. Of course, optimized pcode files may also be used as input.

The code generator program is stored as a command file and therefore may be executed simply by typing **CODEGEN** from the top level of the operating system (that is, from **TRSDOS Ready** if you are using TRSDOS). Like the compiler and the optimizer, it has two forms: a short form and a long form.

> NOTE: The file **CODEINIT/DAT** must be on a disk at the computer when you use **CODEGEN. CODEGEN** uses this file for initilization.

<u>The short form:</u>

> **CODEGEN filename**
>
> "**filename**" may include a drive specifier but should not include an extension. Example: **BENCHMARK:2.** The code generator assumes that the extension is **/OBJ.** For this reason, if you are generating code for an optimized program, you should use the long form of the command.
>
> The output of the code generator is placed in a file of the same name, but with the extension **/COD.** If a drive was specified when the code generator was invoked, the **/COD** file is placed on the same drive as the **/OBJ** file. Otherwise, the operating system decides which drive to use.
>
> The **/COD** file that is the output of the code generator may be used just as any **/OBJ** or **/OPT** file, in conjunction with the **RUN** or **LINKLOAD** commands. However, do not attempt to optimize a **/COD** file. The **/COD** files contain machine instructions and the optimizer accepts only pcode instructions.

The long form:

```
CODEGEN
INP_OBJ = inputfile/ext
OUT_COD = outputfile/ext
DO YOU WANT ASSEMBLY LANGUAGE SOURCE? (Y/N): y or n
```

The long form requires that you enter the full file name, including extensions, for both input and output files. Filenames may also include drive specifiers (example, **BENCHMARK/OBJ:2**). If assembly language output is desired, answer **Y** to the last prompt. Otherwise, answer **N**. If assembly language output is requested, the following prompt will appear: **SOURCE = file/ext**

The additional assembly language output will be directed to the file specified.

6.c. Example Use of the Code Generator

The following is an example of generating code for a program called **BENCHMK.**

step 1 compile **BENCHMK/PCL**

command: **PASCAL BENCHMK/PCL**

step 2 generate code for the compiled program

command (short form example):

CODEGEN BENCHMK

The above example causes **BENCHMK/OBJ** to be used as input, and directs output to a file called **BENCHMK/COD.**

Step 2, using the long form to invoke the code generator, might be:

```
CODEGEN
INP_OBJ  = BENCHMK/OBJ
OUT_COD  = BENCHMK/COD
DO YOU WANT ASSEMBLY LANGUAGE SOURCE ? (Y,N): N
```

The above example does exactly the same thing as the short form example. The example below does the same thing as the previous examples except that it also generates an assembly language output which is directed to the file **BENCHMK/SRC.**

```
CODEGEN
INP_OBJ  = BENCHMK/OBJ
OUT_COD  = BENCHMK/COD
DO YOU WANT ASSEMBLY LANGUAGE SOURCE ? (Y,N): Y
SOURCE   = BENCHMK/SRC
```

7. MIXED MODE OPERATION

Through the use of the linking loader (**LINKLOAD**), pure pcode object (**/OBJ**) files may be linked with code-gened (**/COD**) files. Executable programs (**/CMD** files) may then be built which contain mixed instructions, both pcode and machine code. This ability is important when you are writing large programs. It allows you to select and code-gen only those parts of a program which most affect the speed of execution. The remaining parts of the program can be left in pcode form. This mixed mode operation allows you to increase execution speed without dramatically increasing program size.

7.a. When to Use Mixed Mode

The use of mixed mode is usually not important until you start developing large programs. Small programs can be totally code-gened without the size increase becoming a significant factor. However, completely code-gening large programs (>1000 lines) may cause a size increase which will prevent the program from being executed. The code size of the program can become so large that there is no longer enough room for data storage. This of course depends on the data storage requirements of the program.

When developing large programs, you should not consider code generation until after you have executed the program in pcode form. Observe the execution speed to determine whether or not it is adequate for your application. If not, the next step is to decide what areas of the program are most affecting the speed. Long loops are typical areas in the program where most of the execution time is spent. Another area might be a low level procedure or several procedures which are called frequently throughout the program. After deciding which areas of the program are affecting execution speed the most, separate them from the rest of the program and code-gen them.

The selection and separation process is easiest if the program is well modularized. That is, the program is already segmented into modules, each performing a distinct and well-defined function.

7.b. How to Use Mixed Mode

Once particular areas of a program have been selected for code generation, they must be separated from the rest of the program. Any selected modules (procedures and/or functions) should be declared as externals. (How to declare externals is discussed in Chapter Nine of the Language Reference Guide.)

These separate areas should then be compiled separately from the rest of the program. The compiler **$NULLBODY** option is required to compile procedures and/or functions which are separated from the main program body. Once compiled, the selected areas may be code-gened and then linked to the remainder of the program using the linking loader. Once linked, a command file can be built using the **BUILD** command of the linking loader.

NOTE: There is an alternative way of separating modules of a program without separating them in the Pascal source. The pcode object (**/OBJ**) file can be split (see page F-28).

7.c. Example of Mixed Mode Operation

The process of mixed-mode operation is summarized in the following list of steps.

1. Select the areas which most affect program speed.

2. Separate the selected parts of the program from the remainder of the program. Any selected procedures or functions should be declared as **EXTERNAL** in the main program. Place the separated modules in a separate file or files. Use the **$NULLBODY** compiler option to put the separated modules in a form suitable for the compiler.

3. Compile all parts of the program.

4. **CODEGEN** the parts of the program that were selected to increase execution speed.

5. **LINKLOAD** all compiled parts of the program together and build an executable command file.

The example below demonstrates this process. The program used for this demonstration (listed on page F-20) does not perform any useful function, but merely demonstrates mixed mode operation. In reality, a program of the size demonstrated should be totally translated to machine instructions rather than using mixed mode. The size increase due to code generation is insignificant for such small programs.

NOTE: If you want to follow along with this demonstration, begin by making a diskette that has plenty of storage space. See Appendix III for details.

Step 1: Select the areas affecting execution speed the most.

Examining the program, you can see that the procedure named **LOOP** contains a very long **FOR** loop (1 to 10000). Inside this loop is a long calculation. Any long loop containing a significant number of statements or calculations will benefit substantially from code generation. The procedure **LOOP** is where the majority of program execution time is spent. Therefore, it is a good choice for code generation.

```
(*$NO INOUT*)
PROGRAM MIXED_MODE;
TYPE    ALPHA  = ARRAY(.1..8.) OF CHAR;
        FILENM = ARRAY(.1..72.) OF CHAR;
VAR      FN : FILENAME;
        ID,T : ALPHA;
        OUTPUT : TEXT;

PROCEDURE TIME(VAR T : ALPHA); EXTERNAL;
PROCEDURE SET$ACNM(F : TEXT; VAR FN : FILENM; LEN : INTEGER;
                ID : ALPHA); EXTERNAL;

PROCEDURE LOOP;
(* MODULE TO BE CODEGENED *)
VAR CALCULATION : INTEGER;
BEGIN
    FOR I:=1 TO 10000 DO
    BEGIN
    CALCULATION:=1+2+3+4+5+6+7+8+9+10+11+12+13+14+15
    END
END;  (* END LOOP *)
BEGIN            (* MAIN PROGRAM *)
    (* DIRECT OUTPUT TO THE SCREEN *)
    FN(.1.):=':'; FN(.2.):='C';
    ID:='OUTPUT  ';
    SET$ACNM(OUTPUT,FN,2,ID);
    REWRITE(OUTPUT);
    TIME(T);
    WRITELN(OUTPUT,'STARTING TIME : ', T);
    LOOP;
    TIME(T);
    WRITELN(OUTPUT,'FINISHING TIME : ', T);
END.         (* END PROGRAM *)
```

Step 2: Separate the selected modules from the rest of the program, declaring them as externals in the main program and putting them into a form suitable for compiling.

The following listing shows the procedure **LOOP** separated from the main program. It is declared as an external procedure within the main program and the compiler **$NULLBODY** option is used to turn the procedure into a valid Pascal program. The main program and the procedure **LOOP** must be placed in separate files. For example, the main program could be placed in a file named **MAIN/PCL** and the procedure placed in a file named **LOOP/PCL**.

```
(*$NO INOUT*)
PROGRAM MIXED_MODE;
TYPE    ALPHA  = ARRAY(.1..8.) OF CHAR;
        FILENM = ARRAY(.1..72.) OF CHAR;
VAR      FN : FILENAME;
        ID,T : ALPHA;
        OUTPUT : TEXT;

PROCEDURE TIME(VAR T : ALPHA); EXTERNAL;
PROCEDURE SET$ACNM(F : TEXT; VAR FN : FILENM; LEN : INTEGER;
                ID : ALPHA); EXTERNAL;

PROCEDURE LOOP; EXTERNAL;
BEGIN              (* MAIN PROGRAM *)
    (* DIRECT OUTPUT TO THE SCREEN *)
    FN(.1.):=':'; FN(.2.):='C';
    ID:='OUTPUT  ';
    SET$ACNM(OUTPUT,FN,2,ID);
    REWRITE(OUTPUT);
    TIME(T);
    WRITELN(OUTPUT,'STARTING TIME : ', T);
    LOOP;
    TIME(T);
    WRITELN(OUTPUT,'FINISHING TIME : ', T);
END.         (* END PROGRAM *)
```

```
PROGRAM SEPARATE_COMPILATION;
PROCEDURE LOOP;
(* MODULE TO BE CODEGENED *)
VAR CALCULATION : INTEGER;
BEGIN
    FOR I:=1 TO 10000 DO
    BEGIN
    CALCULATION:=1+2+3+4+5+6+7+8+9+10+11+12+13+14+15
    END
END;  (* END LOOP *)

BEGIN        (* MAIN PROGRAM *)
    (*$NULLBODY*)
END.
```

Step Three: Compile all parts of the program.

```
TRSDOS Ready
PASCAL MAIN

    TRS-80 PASCAL   VERSION: 2.0  00:01:28  05/03/82     PAGE    1
    (*$NO INOUT*)
    PROGRAM MIXED_MODE;
    TYPE ALPHA   = ARRAY(.1..8.) OF CHAR;
         FILENM = ARRAY(.1..72.) OF CHAR;
    VAR       FN : FILENAME;
         ID,T : ALPHA;
         OUTPUT : TEXT;

    PROCEDURE TIME(VAR T : ALPHA); EXTERNAL;
    PROCEDURE SET$ACNM(F : TEXT; VAR FN : FILENM; LEN : INTEGER;
                           ID : ALPHA); EXTERNAL;

    PROCEDURE LOOP; EXTERNAL;

    BEGIN               (* MAIN PROGRAM *)
         (* DIRECT OUTPUT TO THE SCREEN *)
         FN(.1.):=':'; FN(.2.):='C';
         ID:='OUTPUT  ';
         SET$ACNM(OUTPUT,FN,2,ID);
         REWRITE(OUTPUT);
         TIME(T);
         WRITELN(OUTPUT,'STARTING TIME : ', T);
         LOOP;
         TIME(T);
         WRITELN(OUTPUT,'FINISHING TIME : ', T);
    END.           (* END PROGRAM *)
NO ERRORS DETECTED

TRSDOS Ready
PASCAL LOOP

 TRS-80 PASCAL   VERSION: 2.0    00:03:34  05/{03/82     PAGE   1

    PROGRAM SEPARATE_COMPILATION;
    PROCEDURE LOOP;
    (* MODULE TO BE CODEGENED *)
    VAR CALCULATION : INTEGER;
    BEGIN
         FOR I:=1 TO 10000 DO
                BEGIN
                CALCULATION:=1+2+3+4+5+6+7+8+9+10+11+12+13+14+15
                END
    END;  (* END LOOP *)
    BEGIN          (* MAIN PROGRAM *)
         (*$NULLBODY*)
    END.

NO ERRORS DETECTED
```

Step Four: CODEGEN the parts selected to increase speed.

```
TRSDOS Ready
CODEGEN LOOP
LOOP

STACK USED = 15915 of 17344  HEAP USED = 882 of 3756
```

Step Five: LINKLOAD all compiled parts of the program and build an executable command file.

```
TRSDOS Ready
LINKLOAD
L=LOAD, R=RUN, T=TRSDOS, I=INIT, S=SYMBOLS, B=BUILD CMD
>> L
FILE     = MAIN/OBJ
MIXED_MO
32239 BYTES LEFT
>> L
FILE     = TRSLIB/OBJ
SETCSR
GOTOXY
GETKEY
INKEY
CLEARSCR
CLEARGRA
WRITECH
INP
GET$PROC
IO$ERROR
HP$ERROR
TIME
DATE
ITIME
SETPOINT
RSETPOIN
TESTPOIN
USER
CALL$
$MEMORY
NOBLANK
READCURS
PEEK
POKE
INIT$FIL
FILE$STA
SET$ACNM
30669 BYTES LEFT
>> B
STACK SIZE:
FILE     = MIXED/CMD
```

The program, stored under the filename **MIXED/CMD**, may now be executed by typing and entering **MIXED**.

```
TRSDOS Ready
MIXED
STARTING TIME : 00:02:48
FINISHING TIME : 00:02:53
```

The execution time spent inside the **LOOP** procedure may be calculated by subtracting the starting time from the finishing time. With the **LOOP** procedure code-gened, execution time is saved. To find out the exact time difference, use **LINKLOAD** again, this time linking **LOOP/OBJ** instead of **LOOP/COD**.

8. TECHNICAL OVERVIEW OF THE SYSTEM

8.a. The Compiler

The Model III Pascal compiler for the stand-alone Model III or Model 4 is an 8500 line Pascal program which has itself been compiled into a very compact pcode form. The pcode form of the compiler has further been reduced in size by the optimizer. Optimization was necessary in order to make the compiler run in a 48K system. The pcode form of the compiler was reduced in size by approximately 28% from 39K down to 28K.

8.b. The Pcode

The pcode generated by the compiler was specifically designed for the Pascal language. The pcode resembles the assembly language for a stack machine. The pcode was designed to efficiently implement Pascal functions. Therefore, each pcode instruction performs a much more complex function than a machine instruction. In fact, a pcode instruction is equivalent to an assembly language subroutine. This is the reason that pcode is so much more compact than native machine code.

8.c. The Interpreter

The interpreter is a highly optimized assembly language program whose purpose is to interpret pcodes. Since the computer hardware cannot understand pcode instructions, the interpreter is necessary to execute programs which have been compiled into pcode instructions. The interpreter can be thought of as a processor whose instruction set is the set of pcodes.

The interpreter has the ability to switch between pcode and machine code. A particular pcode instruction tells the interpreter that native machine instructions follow. The interpreter then points the program counter (PC) register to the first of the native machine instructions and the hardware begins executing instructions. The ability of the interpreter to switch between pcode and machine code allows programs to contain mixed instructions. This means that parts of a program may be code-gened for speed while the remainder of the program is left in pcode form for compactness.

8.d. The Runtime Support

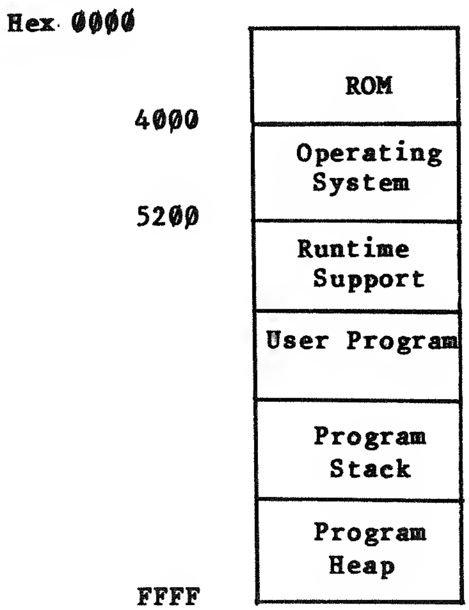
The runtime support consists of the interpreter, a loader, a routine to set up the Pascal stack and heap, and all the input/output (I/O) routines. When building command files with the linking loader, all the runtime support is included with the program being built. Therefore, the total size of an executable program is determined by adding the size of the runtime to the size of the object program. The object program may be pcode, native code, or a mixture of both. The size of the object also includes any libraries which are linked, such as the string library.

8.e. The Memory Map

The following diagram shows the layout of memory usage by the Pascal system. The runtime area is approximately 10K bytes long. The memory remaining after subtracting off the operating system, the runtime, and the program area, is allocated to stack and heap. This is the data area for the program.

The stack is used for storing the program's static variables. The size of the stack is specified at the time the program is run using the **RUN** command, or built using the **LINKLOAD** command. The remainder of memory is allocated to the heap, which is used for storing dynamic variables.

TRS-80



8.f. How the Optimizer Works

The optimizer is a program which contains a loader for loading pcode object files. The loader loads and operates on one module (procedure or function) at a time, maintaining context as it operates on each individual module The pcode instructions are analyzed to determine whether or not they may be compressed into shorter instructions.

Since the compiler is one pass, it must generate some branch and addressing instructions without knowing the actual displacements. This makes it necessary to allocate two-byte operands for unknown displacements in order to handle all cases. However, in many cases, the displacements can be specified using one byte. The optimizer looks for such cases and compresses the pcode instructions in order to take advantage of the need for only a single byte operand.

The optimizer also looks for other types of situations where compression of instructions is possible. For example, all "multiply by two" instructions are converted to "add" instructions. In certain cases, consecutive instructions can cancel one another out (for example, an increment followed by a decrement). The optimizer eliminates such cases. The optimizer also performs constant folding (that is, it replaces arithmetic operations involving only constant operands with a single constant value). For example, 2+2 would be replaced by the single constant 4.

8.g. How the Code Generator Works

The code generator is a program which contains a loader for loading pcode object files. The code generator loads one module (procedure or function) at a time and translates the pcode instructions to machine instructions. As noted earlier, a pcode instruction is equivalent to several machine instructions, so the translation process will increase the total number of instructions in the object (/COD) file.

There are a few pcode instructions which perform very complex functions. To perform equivalent functions in machine code would require a very large number of instructions. Therefore, a few selected pcode instructions are not translated to machine instructions. They are left in pcode form and executed as subroutine calls to assembly language routines within the interpreter. Handling complex functions in this manner prevents the /COD file from becoming as large as it would with complete translation.

9. FOR ASSEMBLY LANGUAGE PROGRAMMERS: INFORMATION ABOUT SYSTEM OUTPUT

9.a. Assembly Language Produced by CODEGEN

The native code generator has the capability of producing assembly language source code in addition to object code. It is not necessary in normal circumstances to assemble the source, since the object code emitted by **CODEGEN** is exactly equivalent to the result of assembling the source. The assembly language is provided as a means for the programmer to examine the code produced by the native code generator. In some cases, the programmer may wish to optimize the code by hand and assemble it. It is expected that the need to do this will be rare, since the effort is substantial and the improvements that can be made are minor. If you wish to assemble the source output of **CODEGEN**, then the Alcor Systems multiprocessor assembler is required.

The source output of **CODEGEN** is useful to the assembly language programmer who wishes to link assembly language modules to Pascal and to call them as Pascal procedures or functions. A possible technique to accomplish this is to write a Pascal procedure or function with the same name and calling sequence as the assembly language routine. The actual code can be left out and accessed by a template that merely accesses the parameters that will be used in assembly language.

The dummy procedure produced above can be compiled by Pascal and run through the code generator with the source option enabled. Pascal and **CODEGEN** will generate the proper Pascal procedure or function linkage and will calculate the addresses of variables and parameters referenced in the body of the procedure. The generated code can then be used as a skeleton for the assembly language that actually implements the functions required.

9.b. CODEGEN Assembly Language Structure

The assembly language code emitted by **CODEGEN** is designed for assembly by the Alcor Systems multiprocessor assembler. This assembler provides the features required to support Pascal and the ability to mix Z80 code (or 6502 code or 1802 code or 8080 code) with Pcode. Essential assembler features include the ability to switch among target processors (Z80 to pcode), the ability to define and reference external symbols (externals are resolved at link edit or load time) and the ability to generate pcode addressing modes (program counter relative, stack displacement, access to common blocks).

Each Pascal procedure or function forms a separate module. All symbols, labels, and instructions are local to the module and reference other modules only through explicit external references. Modules begin with a module identification. For Pascal generated code, the module name is the name of the procedure or function truncated to 8 characters. Each procedure or function also contains an external definition of the procedure name. This is signaled with the "**DEF**" assembler directive. The **DEF** statement causes the name and its value to be defined externally so that other modules can call it.

Switching between modes (native vs. pcode) takes place within the procedure. Some operations performed by Pascal are sufficiently complex that they are implemented with subroutines. Inclusion of the actual code in-line would make the generated code unreasonably large. When these operations (such as input, output, or set operations) are performed, the code generator produces a call to a runtime procedure. These runtime procedures are already part of the pcode interpreter. Rather than reference them again (and require another copy) the processor is switched back to pcode mode and the interpreter is allowed to perform the operations.

When in mixed mode, all procedure calling is performed using the pcode interpreter. Since code for each module is separate, and since modules may be split before being loaded, it is unknown whether the procedure being called is pcode or native code. Therefore, every module is entered in pcode mode. If the module is native code, the processor is switched to native mode immediately after entry to the procedure.

9.c. CODEGEN Assembly Language Format

The native code emitted by **CODEGEN** uses extended 8080 mnemonics. This is done primarily for historical reasons and since the 8080 instruction set more clearly distinguishes instructions by format. Use of 8080 extended mnemonics affects only the source output of **CODEGEN**, as the Z80 instruction set is used and converted directly to object code by **CODEGEN**. Each instruction occupies one line. Labels are left-justified and begin with a letter. Each instruction has an op code which is either an 8080 instruction or a Z80 instruction. There are also pseudo-operators (pseudo-ops) that provide instructions to the assembler rather than generating code.

Operands use standard register names. In many cases, the names of the Z80 index registers are merged with the opcode (for example, **PUSHIX** pushes the **IX** index register). This simplifies interpretation by the assembler. Operands may also use symbolic labels and constants. Constants are normally expressed in hexadecimal (base 16) with a leading greater-than sign (>) to specify hexadecimal to the assembler.

9.d. List of Pseudo-Ops

IDT identifies the module and gives it a name

EQU defines the value of the label to the result of evaluating the operand

DEF defines the operand as an external symbol

REF specifies that the operand is an external symbol that is defined in another module

CSEG specifies the name and size of a common block

QLIST selects the compact format for the assembler listing

END	signals the end of the module
ENTRY	defines an entry point into the module
SETCPU	selects the processor whose assembly language is being assembled

9.e. Object Format

TRS-80 Pascal uses its own format for the object code. The main reason for this is that support for many of the features of TRS-80 Pascal are not present in existing object formats. For example, TRS-80 Pascal supports common blocks for statistically allocated variable storage and the object format must in turn allow for this.

The pcode generated by the compiler is address independent. That is, it contains no absolute memory addresses and can execute without change when loaded anywhere in memory. All branching and calling of procedures within the pcode is done relative to the current program counter. Since procedures are compiled into separate modules, calculation of these relative addresses must be done when the code is loaded. The object format supports external references that are program counter relative.

The object code is tagged hexadecimal and is emitted in a line-oriented stream that is compatible with a Pascal text file. In particular, the object code is character oriented and contains only printable ASCII characters. This allows the object to be manipulated by text editors or transmitted over modems. This is not possible with bit oriented formats.

Each item in the object file begins with a tag which is usually an upper-case letter. The tag defines the type of item and the number and size of the fields to follow. Tags are followed by one or more fields that specify the information to be loaded. Three types of fields exist. Bytes are specified with a two-character hexadecimal number. Words consist of a four-character hex number with the most significant byte first. Labels consist of eight-character names that are the names of external symbols, common blocks, etc.

Following is a table which lists all the tags used in an object file. All tags are followed by one to three fields of information, each field being either a byte, word, or label. The meaning of each tag is also shown.

Tag	Field1	Field2	Field3	Meaning
A	byte			Absolute (non-relocatable) byte
E				End of module
F				End of line
G	word	label		Definition of external symbol
I	word	label		External reference declaration
J	label			Module name
Q	word			Reference to external symbol
M	word	word	label	Definition of common block

Tag	Field1	Field2	Field3	Meaning
N	word			Reference to common block
O	word	word		Overlay definition
P	word			Code (PC) origin
K	word			Relative reference to external
W	word			Relocatable word
X	word			Absolute word
Y	word			Entry point definition
:				End of file

9.f. Splitting Object Modules

Since object files are in ASCII format, they may be edited with a text editor or used as input to a Pascal program. The following is a list of the pure pcode output (**/OBJ**) file for the **LOOP** procedure in the mixed mode operation example. Following it is a listing of the object (**/COD**) file which results from running the pcode object through the code generator. As you can see, the code generation has caused the code to approximately double in size.

<u>Pure pcode listing (/OBJ):</u>

```
JLOOP     P0000G0000LOOP     A01X0000A38A02A03X0001A15A04A10A04A03X2710A07F
A15A06A2BA4EX0000A03X0001A03X0002A22A03X0003A22A03X0004A22A03X0005A22A03F
X00006A22A03X0007A22A03X0008A22A03X0009A22A03X000AA22A03X000BA22A03X000CF
A22A03X000DA232A03X000EA22A03X000FA22A15A02A10A04A30X0004A10A06A27A21AB9F
P00014X0047P005DA3AP0001X0006E
```

<u>Native code listing:</u>

```
JLOOP     G0003LOOP     AC1AEBAE9A01X0006A38A02A55A21X0001ADDA75A04ADDA74F
A05ADDA6EA04ADDA66A05AE5A21X2710AE5ADDA75A06ADDA74A07AC1AE1A78AACAEDA42F
A28A09A47A3FA1FAA8A07AE6A01A18A02A3EA00AA7AC2X0000A21X0001AE5A21X0002AC1F
A09AE5A21X0003AC1A09AE5A21X0004AC1A09AE5A21X0005AC1A09AE5A21X0006AC1A09F
AE5A21X0007AC1A09AE5A21X0008AC1A00AE5A21X0009AC1A09AE5A21X000AAC1A09AE5F
A21X000BAC1A09AE5A21X000CAC1A09AE5A21X000DAC1A09AE5A21X000EAC1A09AE5A21F
X000FAC1A09ADDA75A02ADDA74A03ADDA6EA04ADDA66A05AE5ADDAE5AE1A01X0004A09F
A4EA23A46A03A70A2BA71ADDA6EA06ADDA66A07AC1AAFAEDA42A20A01A3CAA7ACAW003AF
P0038W00BDP00BDACDW0000A3AE
:
```

Each module in an object file begins with the module name. Therefore it is possible to split a file containing several modules into several files, each containing one module. This is an alternate method of segmenting large programs where it is desired to perform code generation on only selected parts.

There are two ways to split the object modules. One is to text edit them. The other more desirable method is to write a Pascal program to split them. A simple program may be written to read the pcode (/**OBJ**) file. Each time a module is encountered, open a file of the same name as the module and write the module to that file. Once all the modules are separated into different files, selected modules may be input to the code generator and translated to native machine instructions. The linking loader may then be used to link the individual modules and build an executable command (/**CMD**) file.

APPENDIX I:

GENERAL LOADING INSTRUCTIONS FOR MODEL III/4

NOTE: Before doing anything else, you should make a backup copy of the Model III Pascal disks that came with this package. Directions for making backup copies begin on page A-12.

Two-Drive System

1. Turn on the computer. The on/off switch is located under the right edge of the keyboard.

2. When the red light goes off, insert the Model III Pascal diskette that you want to use into Drive Ø (the bottom drive next to the video screen), with the square notch to the left and the label facing up. Then close the disk drive door.

3. Insert the other Model III Pascal diskette (OR insert a formatted diskette or a TRSDOS diskette, to be used for storing student programs) into Drive 1. Close the disk drive door. (Instructions for making a formatted diskette appear on page F-37.)

4. Press the orange Reset button.

5. When you see the message **Enter Date MM/DD/YY**, type the date, using two digits each for the month, day, and year. (Example: **Ø2/22/85** for February 22, 1985.) Then press **ENTER.**

6. When you see the message **Enter Time HH:MM:SS**, press **ENTER** to skip the time.

7. When you see **TRSDOS Ready**, the computer is ready to receive a command. This command will probably be the name of a Pascal program. For example, you'd type and enter **ED/CMD** or simply **ED** to use the Pascal editor.

One-Drive System

Follow steps 1-2 and 4-7 above. One-drive users should read Appendix III to learn how to create space for storing Pascal programs.

APPENDIX II:

THE LDOS PATCH

NOTE: The following information applies only to TRS-80 Pascal for a non-network Model III or Model 4 system. Network Pascal does not require that any patches be made -- even if you are using the hard disk.

TRSDOS USERS:

- If you intend to execute the TRS-80 Pascal system with the supplied TRSDOS 1.3 operating system on a Model III or 4 system, you may skip the following pages on patching the system.

LDOS USERS:

- If you intend to execute TRS-80 Pascal using the LDOS operating system Model III/4 floppy or hard disk versions, you must patch the TRS-80 PASCAL system for proper execution. Failure to do so will cause unpredictable results. All patching should be performed while using TRSDOS (LDOS not supplied).

  Note that "LDOS" in this manual refers to the floppy-based version 5.1 and the Radio Shack Hard Disk Operating System for stand-alone Model III computers (or for Model 4 in Model III mode).

HARD DISK USERS:

- Once the Pascal files have been patched for execution under LDOS, they may be copied to the hard disk drive just like any other user file. For further details, see the Hard Disk Operating System manual. Do not copy the TRSDOS system files to the hard disk drive.

A. THE PATCH PROCESS

The TRS-80 Pascal system includes a program for patching disk files. This program is used to apply all current and future patches to the Pascal system. All patches to TRS-80 Pascal files should be applied with the PATCHER program since it contains extensive error checking to assure that patches have been applied correctly. On the next page is an outline of the patch process.

WARNING: NEVER APPLY PATCHES TO THE ORIGINAL DISKETTES. MAKE A BACKUP COPY AND APPLY PATCHES TO THE COPY.

1. Make a backup copy of the release diskette. Do not apply patches to the original release diskette.

2. The text of the patches should be entered into a file using the text editor if in printed form.

3. Load the diskette containing the file named **PATCHER/CMD** and type: **PATCHER.** After the program has been loaded, the diskette containing the program may be removed.

4. The program will prompt for the drive number to be used for the patching process. All diskettes that are to be patched must be inserted into this drive during the patch process.

5. Enter the name of the file or device to be used for the listing. The patch program will echo a patch file listing to this file, and will display any error messages there. Instead of a filename, **:L** may be entered. This causes the listing to go to the printer.

6. Enter the name of the file containing the patches. This is known as the patch control file. The patch control file must remain on-line during the entire patching process. On two drive systems, the patch control file should be on the system diskette. (The patch control file required to modify TRS-80 Pascal for operation with LDOS and the Model III is on TRS-80 Pascal Disk One in a file named **LDOS/PAT.**)

7. Change diskettes when prompted to do so. If any errors are detected, error messages will be displayed and the patches will not be applied. If a patch file requires diskettes to be changed, and they are not, error messages will be generated for those files not present on the diskette in the patch drive. They may be ignored if those files are not required to be patched.

NOTE: Once a file has been successfully patched, it may not be patched again using the same patch control file. This is because all patch control files contain information about the text that was previously in the file to be patched. Once the file has been altered, then that information is no longer valid. If you are not sure that the patches have been applied properly, make a backup of the master diskette and re-apply the patches using the patch control file.

B. EXAMPLE PATCH SESSION

The following is an example of how to patch the Pascal system for execution under LDOS on the Model III. A two-drive system running TRSDOS 1.3 is assumed. The following steps should be performed before any patching is attempted.

1. Copy **LDOS/PAT** to the system disk.
2. Insert the disk with **PATCHER/CMD** and invoke **PATCHER.**

The following information will be prompted for at the terminal. Text after the ";" are comments in this manual and will not be present in the terminal session.

NOTE: After the patch process is complete, use the LDOS CONV utility to convert the TRSDOS formatted diskettes to LDOS format.

```
ENTER DISK DRIVE FOR PATCHES: 1  ; Use drive number 1
LISTING = :L  ; Echo listing to printer (:D for no device)
ALCOR SYSTEMS DISK PATCH UTILITY 1.0 (C) 1982
PATCHES = LDOS/PAT:0   ; File containing patches on the system disk
LOAD DISK : PASCAL1 INTO DRIVE 1
   PRESS <ENTER> WHEN READY  ; Hit enter to start patching disk1
LOAD DISK : PASCAL2 INTO DRIVE 1
   PRESS <ENTER> WHEN READY  ; Hit enter to start patching disk2
STACK USED = 514 of 4032 HEAP USED = 1574 OF 29832
TRSDOS READY
```

If ":L" was used for the listing device, then the following listing will appear at the printer.

```
ALCOR SYSTEMS DISK PATCH UTILITY 1.0 (C) 1982
;
;  TRS80 MODEL III FOR LDOS
;
F, RUN/CMD, PASCAL1
P,15BC,0578,0001,00,13
P,0DFF,05AC,0001,03,00
W,F4DC
;
F, PASCAL/CMD, PASCAL1
P,134C,054A,0001,00,13
P,0ED1,0554,0001,03,00
W,F562
;
F, ED/CMD, PASCAL1
P,11ED,057D,0001,00,13
P,0BD5,055E,0001,03,00
W,F525
;
F, PATCHER/CMD, PASCAL1
P,0FA1,054E,0001,00,13
P,0B16,0529,0001,03,00
W,F589
```

```
;
F, LINKLOAD/CMD, PASCAL2
P,15BC,0578,0001,00,13
P,0DFF,05AC,0001,03,00
W,F4DC
;
F, PASCALB/CMD, PASCAL2
P.0D5C,056E,0001,00,13
P,1575,051E,0001,03,00
W,F574
;
F, CODEGEN/CMD, PASCAL2
P.111A.0535,0001,00,13
P,0C9F,0583,0001,03,00
W,F548
;
F, OPTIMIZE/CMD, PASCAL2
P,111A,0535,0001,00,13
P,0C9F,0583,0001,03,00
W,F548
E
```

APPENDIX III:

DISKETTE MANAGEMENT -- CREATING MORE SPACE FOR PROGRAM STORAGE

Model III Pascal Disks One and Two do not contain enough free space to store student Pascal programs in addition to the Pascal development programs that were provided on the disks. The information below should help you make a diskette that has sufficient storage space for your programs.

Two-Drive Users

Two-drive users can make a formatted diskette to use in Drive 1 for program storage. Drive Ø would contain Model III Pascal Disk One or Disk Two -- whichever contained the development program currently in use.

A formatted diskette is a diskette that has been prepared to receive data. The formatted diskette does not contain the TRSDOS operating system, and therefore it cannot be used in Drive Ø.

To format a diskette, follow these steps:

1. Turn on the computer.
2. When the red light goes off, insert any TRSDOS diskette (any diskette that contains the TRSDOS operating system) in Drive Ø.
3. Insert a new, blank diskette into Drive 1.
4. Press the orange Reset button.
5. When asked for the date, enter the date as described on page F-31.
6. When asked for the time, press **ENTER**.
7. When **TRSDOS Ready** appears, type **FORMAT** and press **ENTER**.
8. When you see the message **Format Which Drive?**, type 1 and press **ENTER**.
9. When you see the message **Diskette Name?**, type **PROGRAMS** (or type any other name of eight or fewer alphanumeric characters, beginning with a letter), and press **ENTER**.
10. For **Master Password**, type **PASSWORD** and press **ENTER**.

When the formatting process is successfully completed, "**ØØ Flawed Tracks**" will be displayed and **TRSDOS Ready** will reappear. If you see any number except "**ØØ**" for "**Flawed Tracks**", or if an error message appears, use another new, blank diskette and repeat the instructions from step 3.

One-Drive Users

One-drive users cannot use formatted diskettes. But you can use a TRSDOS system diskette that has room both for your programs and for some of the Pascal development system programs. You can do this by copying to a TRSDOS system diskette only those Model III Pascal files that you will be using. (Pages F-1 and F-2 provide details on the purpose of each program on the two Model III Pascal diskettes.)

Individual files may be copied from one disk to another using the TRSDOS **COPY** command. The process is as follows:

1. With a TRSDOS system diskette in the drive and the **TRSDOS Ready** prompt showing, type the following and press **ENTER** (be sure to substitute the filename and extension of the program that you want to copy for the words **filename/ext** below):

   COPY filename/ext:Ø :Ø

2. When you see the message **Insert SOURCE Disk (ENTER)**, make sure that the diskette in the drive is the disk that already contains the program (your SOURCE disk). If not, change disks. Then press **ENTER.**

3. When you see the message **Insert DESTINATION Disk (ENTER)**, remove the SOURCE disk and insert the disk that you want to copy the program onto (your DESTINATION disk). Then press **ENTER.**

4. Continue to switch disks as directed by the computer. When the copying is complete, **TRSDOS Ready** will reappear.

APPENDIX IV:

TROUBLE SHOOTING GUIDE

A. MISCELLANEOUS ERRORS

1. **Problem:** While a file is being edited, the latter part of the file is found to be missing.

   Solution: Use the **APPEND** command to page the latter part of the file into the text buffer. **APPEND** is explained in the Editor Guide.

2. **Problem:** Upon exiting the editor, you see an **IO ERROR** message displayed.

   Cause/Solution: The diskette was full or not in the drive. See the Editor Guide for information on recovering the original file from the **T011/TMP** work file.

3. **Problem:** During a compile, the Pascal compiler abnormally terminates with a **FATAL ERROR - OUT OF HEAP** or **OUT OF STACK** message.

   Cause/Solution: The compiler has insufficient memory space, due to:
   (a) Too large a program (use **PASCALB** version of compiler)
   (b) There are high memory drivers in place that limit the amount of space the compiler has (such as the **KSM** driver in LDOS or a printer spooler). Remove the high memory driver or use **PASCALB.** See page **F-7** for information on invoking the compiler using a stack parameter.

4. **Problem:** When executing your compiled program with the **RUN** command, or when executing a command (**/CMD**) file built with the **LINKLOAD** utility, the program abnormally terminates with the **FATAL ERROR - OUT OF HEAP** or **OUT OF STACK.**

   Solution: Increase the stack specification when invoking the **RUN** command, or increase the stack specification when building the command file with the **LINKLOAD** utility.

5. **Problem:** After you executed the compiler using the long form, where the object and listing files are specified by the user, the original source file suddenly contains object code.

   Cause: You specified the source file as the object file when you invoked the Pascal compiler.

6. **Problem:** When executing under **LDOS**, the **RUN** command is invoked with a Pascal object code file as an argument and the error message **Load file format** error is displayed.

   Cause/Solution: The Pascal **RUN** command must be renamed to **RUNP** or something that does not conflict with the **LDOS RUN** command. You are attempting to execute Pascal object code with the LDOS **RUN** utility.

7. **Problem:** During an edit, you try to insert the external Pascal declarations for the **TRSLIB** library by specifying individual function and procedure names, and nothing is inserted by the editor.

   Solution: All of the declarations are contained in a file named **TRSLIB/PCL.** This file must be inserted to place the declarations into the source file.

8. **Problem:** The compiler continues to compile after the end of the entered source code is encountered. Garbage text is displayed in the listing.

   Solution: The Pascal system files have not been patched for execution under LDOS.

B. COMMON ERROR MESSAGES GENERATED BY THE COMPILER

1. **Missing end** There must be an **END** statement for every **BEGIN** statement in a Pascal program.

2. **Undeclared identifier** All variables must be declared in the variable declaration section. Failure to do so will cause the compiler to generate this message.

3. **Type of parameter does not match formal parameter** An attempt has been made to call a procedure or function using an argument that does not match the formal parameter that was declared in the procedure declaration statement. In special cases, the type matching requirements may be overridden by using the type transfer operator, as described in the Language Reference Guide.

NOTE: See the Language Reference Guide for a complete listing of all error codes generated by the Pascal compiler.

ABS - The predefined Pascal function **ABS** returns the absolute value of the number in its argument. Example: **ABS(-5) = 5.**

alphanumeric character - An "alphanumeric character" is one of the numbers 0-9, or a letter a-z or A-Z.

AND - One of three logical operators in Pascal, **AND** causes two expressions to be evaluated. If both are true, **AND** returns the value "true." If one or both are false, the value "false" is returned. Example: If two Boolean operators, **Saturday** and **zoo**, both held the value "true," then the operation **Saturday AND zoo** would return the value "true."

argument - A piece of information that accompanies a Pascal statement in order to provide data needed to execute the statement is the statement's "argument." In the following example, the argument to the **WRITE** statement tells what to write:

```
WRITE('Testing, testing, 1, 2, 3.');
```

arithmetic operator - An "arithmetic operator" is a symbol that is used in an arithmetic expression to determine how the expression is evaluated. The arithmetic operators in their order of precedence are:

-	negation
*	multiplication
/	real-number division
div	integer division
mod	integer division with the remainder kept as result
+	addition
-	subtraction

ARRAY - The **ARRAY** is a structured data type consisting of a series of variables of a single type. (See Chapter Eight of the TUTORIAL.)

assignment statement - The "assignment statement" is used to assign a particular value to a variable. For example, assuming that the variable **letter** had been properly declared as a variable of type **CHAR**, the following statement would assign the value "**a**" to **letter**:

```
letter := a;
```

BEGIN - **BEGIN** is the word used to mark the start of the program section that contains Pascal statements (you might think of the part **BEGIN** heads as the "active" part of the program).

block-structured language - A "block-structured" language is used to construct programs in a block-like manner, with well-defined sections, rather than writing programs which are linear lists of commands. Pascal allows program blocks to be "nested" such that one block may lie entirely within another block, but blocks do not overlap in any other way.

BOOLEAN - One of the five simple variable types predefined by Pascal. Variables of type **BOOLEAN** may take on either of two values: "true" or "false."

call - To "call" a procedure or function is to access that procedure or function in order to execute it.

carriage return - A "carriage return" ends one line of text and places the cursor at the beginning of the next line. (This term is used in talking about screen or keyboard output or input, or output to a printer.) An example of a carriage return is a press of the **ENTER** key.

CASE - The **CASE** statement is a selection-control statement, used when you need to select for execution one statement from a list of statements. In front of every statement in the list that follows the **CASE** statement is a case-selector constant. The statement whose case-selector constant matches the value in the case-selector variable is the statement that is executed. **END** with a semicolon appears at the end of the list to terminate the **CASE** statement.

CHAR - One of the five simple variable types predefined by Pascal. A variable of type **CHAR** may take on as its value any single character in the ASCII character set (see chart on pages D-93 through D-95).

comment - A "comment" is text that the programmer inserts into a program to explain what the program is doing. Comments inserted between special brace characters ("{" and "}" or "(*" and "*)") are not considered to be part of the program and are ignored by the compiler.

compile - To "compile" a program is to translate it from <u>source</u> code that a programmer can easily read, to <u>object</u> code that a computer or an interpreter program can easily read.

compiler - A program used to translate programs from source to object code.

component field - A "component field" holds one piece of information within a record. For example, an **INTEGER** variable **customernumber** might be a component field of a **RECORD** that contains other component fields such as **customername**.

component variable - A "component variable" can take on the value of a single item within an **ARRAY**. The component variable is of the same type as the array

compound statement - A "compound statement" is a series of statements surrounded by **BEGIN** and **END**. Compound statements are executed as if they formed one statement.

CONST - **CONST** marks the program section in which <u>constants</u> are declared.

constant - A "constant" is an identifier that represents a certain value that never changes (the value remains constant).

declare - To define the value(s) that can be taken on by a variable, data type, or constant.

decoding - The process of translating a variable into a value such as "blue" or "green." Decoding and encoding can often be avoided in Pascal because variables of user-defined data types can be made to take on these values directly.

decrement - To decrease the value of a number by a certain amount.

default - A "default" is a value that is assumed by Pascal if no value is specified by the programmer or user.

DISPOSE - The **DISPOSE** statement releases storage space that was allocated using the **NEW** statement.

document - To "document" a program is to insert comments into it that explain what the program is doing.

dynamic data type - A "dynamic data type" is a data type for which storage is allocated as needed. Chapter Nine of the TUTORIAL and Chapter Five of the LANGUAGE REFERENCE GUIDE provide information on dynamic storage allocation.

END - The statement that marks the end of a Pascal program or program block.

enumerated type - A type is said to be "enumerated" if a list of all possible variable values is given in the type declaration.

execute - The computer "executes" a program or statement when it encounters the program or statement and performs whatever action(s) are directed by the program or statement. To execute a program is to run it.

expression - An "expression" is something that can be evaluated. The following are examples of expressions:

5 + 2 **NOT tuesday** **cost * number**

file **(logical)** - A logical file is an identifier used within a program to route input or output data.

file **(physical)** - A physical file is actual input or output device which the logical file represents. A physical file may be a disk file, the printer, the computer keyboard and screen, or a "dummy" file (in which case no I/O occurs).

FILE OF **<any known type>** - a **"FILE OF <any known type>"** can be declared in Pascal for purposes of storing data to be retrieved at some other time.

flow-control statements - A "flow-control statement" is a statement that is used to alter normal program flow. This may be a statement that causes repeated execution of a statement or set of statements, or it may cause one statement to be executed under certain conditions and some other statement to be executed under certain other conditions.

function - A "function" is a program block whose results must be used (like a variable) in an expression or assignment statement.

global variable - A variable is "global" if it is shared by two or more program blocks.

GOTO - The "**GOTO**" statement transfers control to the statement named in its argument.

heap - The "heap" is the area of memory reserved to contain a program's symbol table while that program is being compiled. During program execution, programs that use pointers and the procedure **NEW** will use storage from the heap. The heap also contains the buffers used to read from and write to files.

input - "Input" is the act of bringing data into a program, and can also refer to the data that is brought in.

increment - To "increment" a number is to increase its value by a certain amount.

INTEGER - One of the five simple data types predefined by Pascal. An "integer" is a number that consists of one or more digits, optionally preceded by a positive or negative sign. No decimal point or fractional portion is allowed.

linked list - A "linked list" is a programming structure in which one dynamically stored data structure contains a variable that points to another such data structure, and so on. (See Chapter Nine of the TUTORIAL.)

local variable - A variable is said to be "local" if it is used only within a single program block.

logical operator - Pascal's logical operators are **AND**, **OR**, and **NOT**.

loop - To loop within a program is to execute a single statement or group of statements repeatedly. The programming structure which sets up this repetition is called a "loop."

modular program - A "modular program" is organized into sections, each of which performs a specific function.

node - The word "node" can be used to refer to a unit of data storage. A single record within a file can be termed a record "node."

NOT - The logical operator **NOT** changes a Boolean value to the opposite value. (**NOT** is to a logical value as a negative sign is to a number.)

null - "Null" means "having no value." A "null" set is an empty set.

object code - Object code is the executable (run-able) program that results when a Pascal source code program is compiled.

one-dimensional array - An array is one-dimensional if it consists of one row of values. An array of this kind is declared as:
ARRAY [1..n] of TYPE
In contrast, a two dimensional array is declared as:
ARRAY [1..n,1..n] of TYPE

ordinal - A data type is said to be "ordinal" if all of its possible values can be arranged in sequence from lesser to greater. For example, a data type that consists of the integers from 1 to 100 is ordinal. The values of an ordinal type need not be numeric, but they do need to be able to be mapped to a sequence of numbers.

output - "Output" is the act of sending data out of a program (for example, to a printer). "Output" can also refer to the data that is sent out.

OR - One of three logical operators in Pascal, **OR** causes two expressions to be evaluated. If either of the expressions is true, **OR** returns the value "true."

PACKED - Any array may be declared with the word **PACKED** as a prefix. **PACKED** tells the computer to store the data elements as efficiently as possible.

parameter list - A "parameter list" is the list of values that are being passed to a procedure.

pass-by-reference variable - A variable is "passed by reference" if it is specifically named in the parameter list and is immediately preceded by the word **VAR**.

pass-by-value variable - A variable is "passed by value" if it is specifically named in the parameter list, but is not immediately preceded by the word **VAR**.

pointer - A "pointer" is a variable that, instead of directly taking on a value, points to the location in memory where a data structure is stored. Pointers are used in dynamically allocating storage to data structures.

procedure - A "procedure" is a Pascal sub-program that performs a specific task within a complete Pascal program.

READ - The **READ** statement causes the program to input data from an outside device (such as the keyboard).

READLN - The **READLN** statement works like the **READ** statement, except that once data has been read, **READLN** sets a marker at the beginning of the line following the line read from -- even if the end of the line was not reached.

REAL - One of the five simple data types predefined by Pascal. A "real" number consists of one or more digits, optionally preceded by a positive or negative sign, and optionally followed by a decimal point and one or more digits.

RECORD - A **RECORD** is a data structure made up of component variables. Examples of the **RECORD** data type appear in Chapter Eight of the TUTORIAL.

relational operator - A "relational operator" is used in comparing two values. Examples of relational operators are: < > = <=

reserved word - A "reserved word" in Pascal is a Pascal language word that has a fixed meaning. These words cannot be used as identifiers. The list of reserved words appears on page D-5.

RESET - **RESET** is used to prepare an input file to be read, as discussed in Chapter Three of the TUTORIAL.

REWRITE - **REWRITE** is used to prepare an output file to receive data, as discussed in Chapter Three of the TUTORIAL.

scope - The boundary within which an identifier can be used is that identifier's "scope." For example, scope rules dictate that an identifier declared in an inner program block cannot be used in an outer program block.

selection control statement - A "selection control statement" causes a specific statement out of a series of statements to be executed. In Pascal, **CASE** is a selection control statement.

set operators - The set operators are " + " (gives the union of two sets), " * " (gives the intersection of two sets), and " - " gives the difference of two sets, as explained in Chapter Ten of the TUTORIAL.

source code - The "source code" of a Pascal program is the program as it is typed in by the programmer, using the editor program. Source code must be compiled before the program can be executed.

statement - A "statement" is a Pascal programming word or phrase that causes the computer to perform a specific action.

static data types - A data type whose storage size is defined before the program is compiled or executed is a "static data type." The size of such data types is fixed.

string - A "string" is a series of characters.

structured data types - A "structured data type" is a data type such as an array or record, that is made up of component variables or fields.

subrange - A data type is said to be a "subrange" of another data type if it consists exclusively of some, but not all, of the members of the other type. A data type encompassing the numbers from 1 to 5 is a subrange of type **INTEGER.**

syntax - "Syntax" refers to the rules that govern how elements of a language must be arranged in order to have any meaning.

TEXT - One of the five simple data types predefined by Pascal. Variables of type **TEXT** are used in routing data for input and output, as described in Chapter Three of the TUTORIAL.

truncate - To "truncate" a number or string of characters is to drop off a certain part of it. This most commonly refers to dropping the fractional portion of a real number, and an integer number is the result of this truncation.

type - The "type" of a variable determines the kind and range of values that the variable may take on. A variable may be of a predefined type (such as **INTEGER**, **REAL**, etc.) or a new type may be declared by the user in the **TYPE** section of the program.

user-defined data type - The programmer may define a new data type by giving the type a name and listing all of the values that the type encompasses. The new type is declared in the **TYPE** section using this format:

newtype = (value1, value2, value3, ... valueN);

VAR - Variables are declared in the **VAR** section of the program using this format:

varname : TYPE;

variable - A "variable" is an identifier that can hold a single value at any given time, and may change value any number of times within the program. A single variable may hold only one kind (or "type") of data.

WRITE - The **WRITE** statement causes the value named in its argument to be output to a logical file. The default file is **OUTPUT**, or a file may be specified.

WRITELN - The **WRITELN** statement works like **WRITE**, except that it causes a line feed after the value is output. The next value that is output after a **WRITELN** statement will begin on a new line.

INDEX

RADIO SHACK A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U. K.
91 KURRAJONG ROAO MOUNT ORUITT, N.S.W. 2770	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAO WEDNESBURY WEST MIOLANDS WS10 7JN